编程狂人

Programming Madman

NO.Christmas

推酷

关于推酷

推酷是专注于 IT 圈的个性化阅读社区。我们利用智能算法，从海量文章资讯中挖掘出高质量的内容，并通过分析用户的阅读偏好，准实时推荐给你最感兴趣的内容。我们推荐的内容包含科技、创业、设计、技术、营销等多方面内容，满足你日常的专业阅读需要。我们针对 IT 人还做了个活动频道，它聚合了 IT 圈最新最全的线上线下活动，使 IT 人能更方便地找到感兴趣的活动信息。

关于周刊

推酷周刊是专为 IT 人打造的行业技术周刊，目前推出的《编程狂人》是献给广大的程序员们。 我们利用技术挖掘出那些高质量的文章，并通过人工加以筛选整理出来。 每期的周刊一般会周一的某个时间点发布。

联系我们


tuicool2012


164644910

推酷网

下载 APP

Android版本

iPhone版本

小酷的话：不知不觉，编程狂人周刊已经连续做 5 期，周刊也是在不断改进，内容不断地在求精，正在努力的做到你们心中 NO.1~想成为你们的依赖！那正好这期也迎来了圣诞节，除了想对大家真诚的说句谢谢外再加一句 Merry Christmas~！

2013/12/23/第 5 期

目录

业界新闻

Objective-Cloud 将 Objective-C 搬到了云端

beego 1.0.0 正式版本发布，Go 应用框架

jQuery Mobile 1.4 正式版发布

NodeJS 无所不能：细数 10 个令人惊讶的 NodeJS 开源项目

前端开发

WEB 设计指南！网页设计师必炼内功技法

JavaScript 跨域总结与解决办法

JavaScript 社区开发者调查：服务端 JS 盛行，Backbone.js 使用最多

CSS2.0 实现面包屑

Grunt-前端利器

跨浏览器开发工作小结

编程语言

聊聊并发（七）——Java 中的阻塞队列

微服务架构解析

为什么 Erlang 比 C 慢那么多倍?

PHP 真正多线程的使用

R 语言教程：写给高级入门者的数据打理攻略

目录

程序设计

独立开发者说：Objective-C 最糟糕的13件事

使用 Objective-C 一年后我对它的看法

对比 iOS 中的四种数据存储

2月1日起提交的应用需针对 iOS 7 优化

朴素贝叶斯分类器的应用

后端架构

SVN 有任何胜过 git 的地方吗?

单元化与分布式架构的切分问题

高性能、高流量 Java Web 站点打造的22条建议

洪强宁介绍 Douban App Engine 的架构与特性

大众点评的大数据实践

MetaQ 在双十二彩票中的运用

Redis 在新浪微博中的应用

RocksDB 介绍：一个比 LevelDB 更彪悍的引擎

目录

程序人生

十大怪异的编程语言

2013 年 -- Facebook 在开源方面的工作介绍

如果老婆和女朋友她们是程序

Reddit 帝国建立在一个有瑕疵的算法之上

项目开发中，你会倾向于质量还是速度?

正文

Objective-Cloud 将 Objective-C 搬到了云端

众所周知，Objective-C 更多用于开发 iOS 和 Mac 上的应用。而日前国外一个新发布的网站实现了将 Objective-C 语言应用搬到了云端，目前处于公测阶段，主要特性：

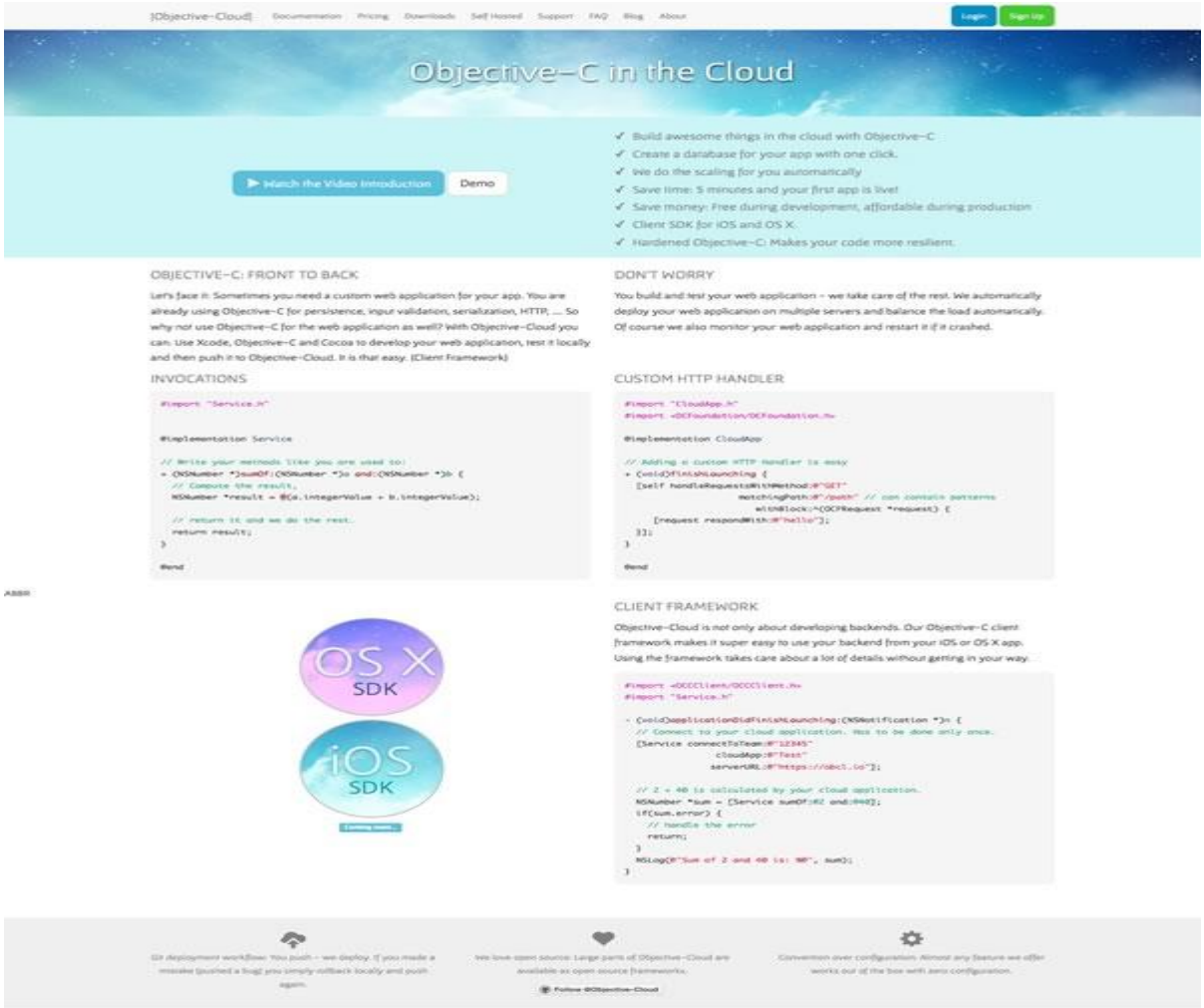
- 可在云端使用 Objective-C 构建完美应用
- 通过简单点击即可创建数据库
- 自动实现伸缩
- 5 分钟内可搞定你的第一个应用
- 省钱：开发免费，产品部署便宜
- 为 iOS 和 OS X 提供客户端 SDK
- Objective-C: 使你的代码更具弹性

下面是一个实现了自定义 HTTP 处理器的示例代码：

```
01 #import "CloudApp.h"
02 #import <OCFoundation/OCFoundation.h>
03
04 @implementation CloudApp
05
06 // Adding a custom HTTP Handler is easy
```

```
07 + (void)finishLaunching {
08     [self handleRequestsWithMethod:@"GET"
0                                                matchingPath:@"/path" //
9  can contain patterns
1                                                        withBlock:^(OCFReque
0  st *request) {
11             [request respondWith:@"hello"];
12     }];
13 }
14
15 @end
```

网站首页（点击访问）



原文

http://www.oschina.net/news/46952/objective-cloud?utm_source=tuicool

beego 1.0.0 正式版本发布 ,Go 应用框架

经过了四个多月的重构开发，beego 终于发布了第一个正式稳定版本。这个版本我们进行了重构，同时针对很多细节进行了改进。下面列一些主要的改进功能：

1. 模块化的设计,现在基本上 beego 做成一个轻量的组装框架,重模块的设计，目前实现了 cache、config、logs、sessions、httplibs、toolbox、orm、context 等八个模块,以后可能会更多。用户可以直接引用这些模块应用于自己的应用中，不仅仅局限于 Web 应用，beego 用户中有应用 logs、config、cache 这些模块到页游、手游中。

2. 工程化的设计，部署项目之后，经常需要进行对线上程序进行各种信息的统计和分析，统计包括 QPS，分析包括 GC、内存使用量、CPU，如果出现问题的时候我们还希望通过 profile 来调试，那么 beego 都为你考虑到了这些，集成了监控模块，默认是关闭的，用户可以开启，并在另一个端口监听，通过 http://127.0.0.1:8088/访问。

3. 详细的文档，这个版本的文档全部是新写的，在之前文档用户的各方面反馈之后，进行了很多细节上面的改进，目前文档中英文版本都已经完成，中英文文档的评论分离，针对不同的用户群交流。

4. 丰富的示例，这一次更新我们开发组写了三个例子，聊天室、短域名、todo 任务三个比较有典型意义的例子。让用户在熟悉 beego 之前有一个更深入的了解。

5. 全新设计的官方网站,这一次我们通过社区获得了很多人的帮助,logo 设计，网站 UI 的改进。

6. 越来越多的用户，官方网站列举了一些典型的用户，都是一些比较大的公司，他们内部都在使用 beego 开发对外的 API 应用，说明 beego 是得到了线上项目验证的框架。

7. 越来越活跃的社区，在 github 上面目前已经差不多有 390 个的 issue，贡献者超过 36 个，commit 超过了 700 个，Google groups 目前还在稳步发展中。

8. 周边产品越来越多，基于 beego 的开源产品也越来越多，例如 cms 系统，

https://github.com/insionng/toropress 例如管理后台系统，

https://github.com/beego/admin

9. beego 的辅助工具越来越强大，bee 工具是专门辅助用户开发 beego 应用的，可以快速的创建应用，动态编译，打包部署等

欢迎各位在使用 beego 过程中玩的愉快，有问题欢迎和我在 github 上面互动，好的框架是靠大家用出来的。

http://beego.me/

相关链接

- beego 的详细介绍：请点这里
- beego 的下载地址：请点这里

原文

http://www.oschina.net/news/47020/beego-1-0-0?utm_source=tuicool

jQuery Mobile 1.4 正式版发布

jQuery Mobile 1.4 正式版发布了，改进记录请看之前发布的 Beta 和 RC 版本。

详细列表请看 ChangeLog

jQuery Mobile（jQueryMobile） 是 jQuery 在手机上和平板设备上的版本。

jQuery Mobile 不仅会给主流移动平台带来 jQuery 核心库，而且会发布一个完整统一的 jQuery 移动 UI 框架。支持全球主流的移动平台。jQuery Mobile 开发团队说：能开发这个项目，我们非常兴奋。移动 Web 太需要一个跨浏览器的框架，让开发人员开发出真正的移动 Web 网站。我们将尽全力去满 足这样的需求。

下载地址：jquery.mobile-1.4.0.zip
原文
http://www.oschina.net/news/47061/jquery-mobile-1-4-final?utm_source=tuicool

NodeJS 无所不能：细数 10 个令人惊讶的 NodeJS 开源项目

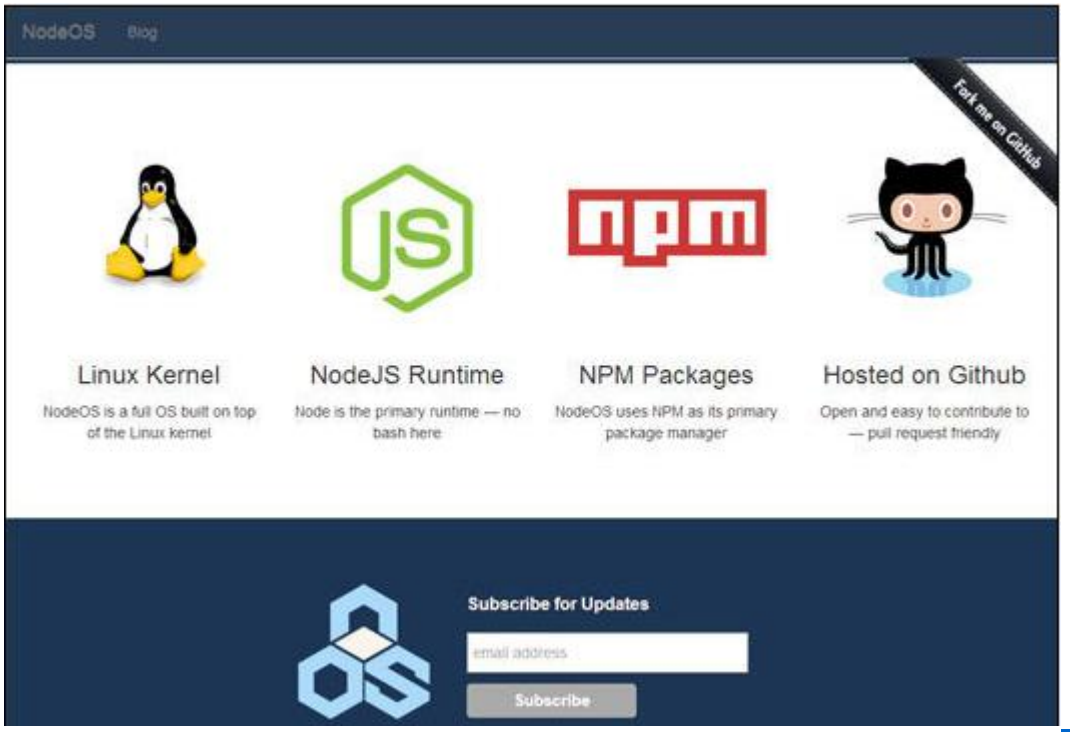
摘要：NodeJS 是一个服务器端 JavaScript 解释器，它将改变服务器应该如何工作的概念。它的目标是帮助程序员构建高度可伸缩的应用程序，编写能够处理数万条同时连接到一个（只有一个）物理机的连接代码。

在几年的时间里，NodeJS 逐渐发展成一个成熟的开发平台，吸引了许多开发者。有许多大型高流量网站都采用 NodeJS 进行开发，像 PayPal，此外，开发人员还可以使用它来开发一些快速移动 Web 框架。

除了 Web 应用外，NodeJS 也被应用在许多方面，本文盘点了 NodeJS 在其它方面所开发的十大令人神奇的项目，这些项目涉及到应用程序监控、媒体流、远程控制、桌面和移动应用等等。

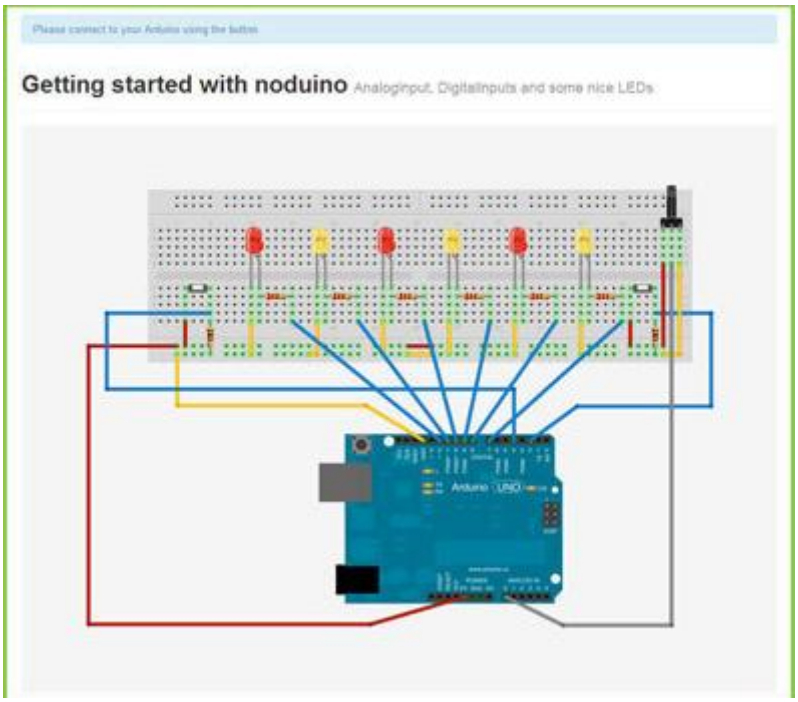
1. NodeOS

NodeOS 是采用 NodeJS 开发的一款友好的操作系统，该操作系统是完全建立在 Linux 内核之上的，并且采用 shell 和 NPM 进行包管理，采用 NodeJS 不仅可以很好地进行包管理，还可以很好的管理脚本、接口等。目前，Docker 和 Vagrant 都是采用 NodeOS 的首个版本进行构建的。



2. Noduino

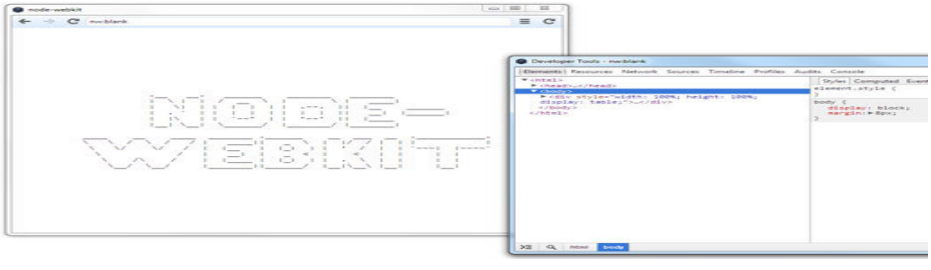
许多硬件黑客希望通过 Web 页面即可控制他们的 Arduino，Noduino 就是这样的一个项目，一个简单灵活的 JavaScript 和 NodeJS 框架，通过使用 HTML5、Socket.IO 和 NodeJS 的 Web 应用来控制 Arduino。目前，该项目刚刚启动，支持一些常用功能，比如从 Arduino 中捕获事件（例如点击按钮）等。



3. Node-WebKit

Node-Webkit 是一个基于 Chromium 与 NodeJS 的应用程序运行器，允许开发者使用 Web 技术编写桌面应用。它是 NodeJS 与 WebKit 技术的融合，提供一个跨 Windows、Linux 平台的客户端应用开发的底层框架。

跨平台开发并非易事，其中一种方式便是使用 Web 技术和 Node-Webkit 开发桌面应用来代替那些庞大且笨重的开发框架。



4. PDFKit

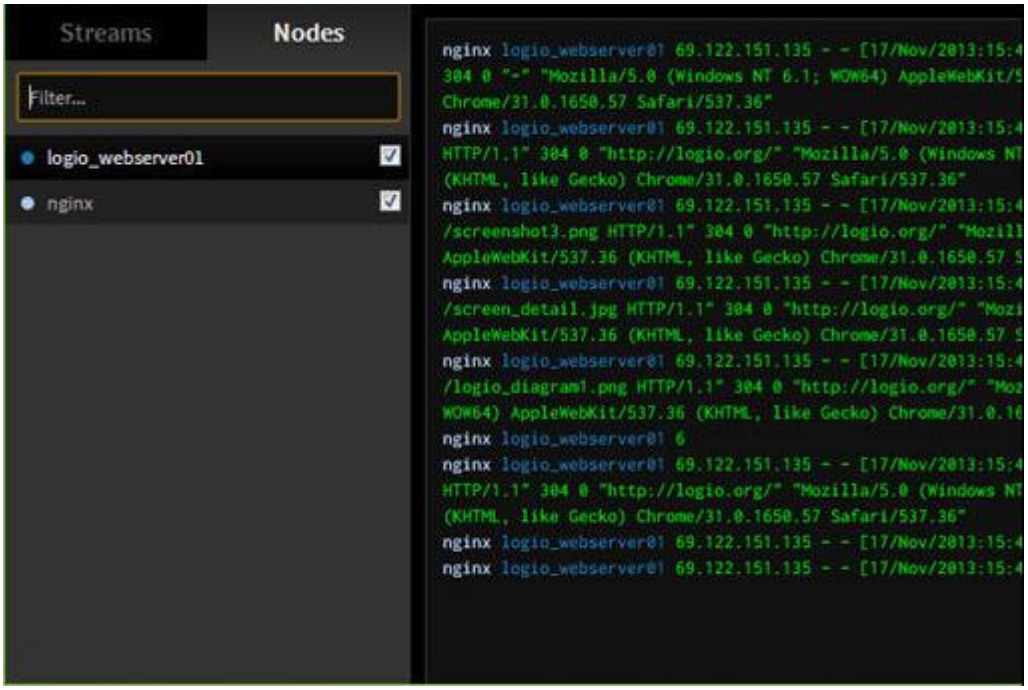
PDFKit 是采用 NodeJS 开发的一款 PDF 文档生成库，它使用一个“HTML5 canvas-like API”来创建矢量图形和字体嵌入，并且支持许多标准的 PDF 功能，如文件的安全性、表的创建、文本换行、项目符号、高亮提示、注释等 PDF 功能。

注意，PDFKit 是一款 PDF 生成工具，而不是一个文档转换系统。如果你想对现有的 PDF 文档进行操作，你可以使用另一个 NodeJS 项目——Scissors。

5. Log.io

Log.io 是一个基于 NodeJS 开发的实时日志监控项目，在浏览器里访问。需要注意的是，Log.io 只监视日志变动并不存储日志，不过这个没关系，只要知道日志存储在哪个机器上。

Log.io 使用 Socket.io 库发送活动报告的，和其他的监控工具一样，Log.io 也采用服务器－客户端的模式。Log.io 由两部分组成：server 和 harveste，server 运行在机器 A（服务器）上监视和纪录其他机器发来的日志消息；log harvester 运行在机器 B（客户端）上用来监听和收集机器 B 上的日志改动，并将改动发送给机器 A，每个需要纪录日志的机器都需要一个 harvester。



6. Nodecast 或 Leapcast

受谷歌 Chromecast 技术的启发，开发者使用 NodeJS 开发出不少 Chromecast 仿真应用。如 Nodecast 或 Leapcast。在 PC 上运行 Nodecast 或 Leapcast，启动移动设备，选择一个支持 Chromecast 的应用程序，然后你就可以把移动广播上的内容映射到电脑上了，把电脑当成一个流媒体使用。

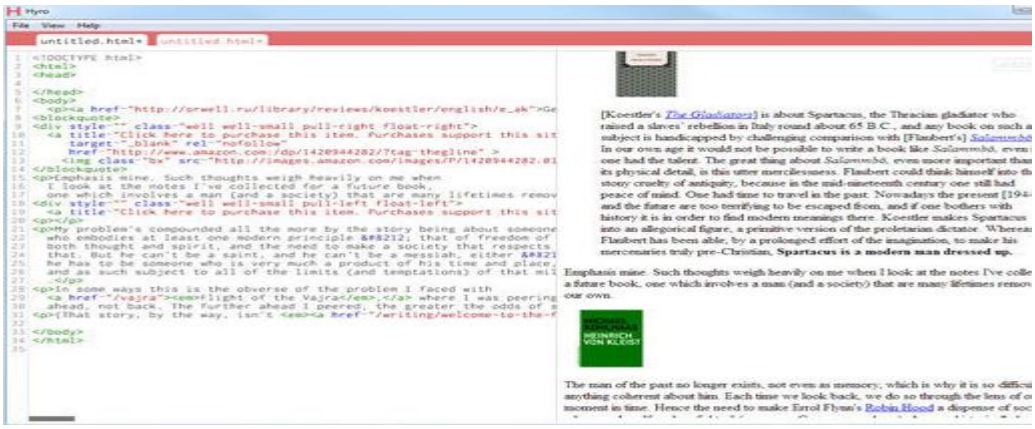
在这两个应用中，Nodecast 比较简单些，但相应的功能也比较少，它仅经过了 YouTube 和 Google Music 的测试(DEMO)。注意,大家不要把 Nodecast 与 Nodecast 库混淆，后者使用 DIAL 发现协议提供链接设备（类似 Chromecast）。

7. Nexe

Nexe 是一款小巧却非常实用的 NodeJS 工具，它可以为 NodeJS 应用创建单一可执行的文件，并且无需安装运行时，这样，一些非技术终端的用户就无需变动 NodeJS 应用的所有依赖程序。如果你想发布一个 NodeJS 应用程序，并且没有 GUI，Nexe 则是您的最佳选择。目前该应用程序的一个弊端是不能在 Windows 平台上工作，只适用于 Linux 和 Mac OS X 平台，并且它也不支持本地 NodeJS 模块。

8. Hyro

Hyro 是使用 NodeJS 开发的一款实时 HTML5 编辑器，如下图所示，左边显示 HTML 源码，右边显示内容。语法高亮由 CodeMirror 提供。Hyro 并不打算成为一款成熟的 Web IDE，更像是一款轻量级的 HTML 或 CSS 记事本。



9. [Haroopad]

Haroopad 是一款 Linux 上的 markdown 编辑器，使用 Chromium 作为 UI，支持 Windows、Mac OS X 和 Linux。主题样式丰富，语法标亮支持 54 种编程语言。 如下图所示，一边是代码编辑窗口，一边是预览窗口，可以实时更新。其邮件导出功能可以将文档发送到 Tumblr 和 Evernote。

10. [TiddlyWiki5]

TiddlyWiki 是一款交互式的 wiki，非常灵活，它也可以在浏览器里作为单一的 HTML 文件或者是一款功能强大的 NodeJS 应用程序。

TiddlyWiki5 是全新设计的 5.0 版本，它可以直接集成 NodeJS 解锁一系列的功能，但在单机模式下是不可用的。目前，TiddlyWiki5 仍处于测试阶段。

原文

http://www.csdn.net/article/2013-12-17/2817827-10-surprising-Node.js-projects?utm_source=tuicool

JavaScript 跨域总结与解决办法

什么是跨域

JavaScript 出于安全方面的考虑，不允许跨域调用其他页面的对象。但在安全限制的同时也给注入 iframe 或是 ajax 应用上带来了不少麻烦。这里把涉及到跨域的一些问题简单地整理一下：

首先什么是跨域，简单地理解就是因为 JavaScript 同源策略的限制，a.com 域名下的 js 无法操作 b.com 或是 c.a.com 域名下的对象。更详细的说明可以看下表：

URL	说明	是否允许通信
http://www.a.com/a.js http://www.a.com/b.js	同一域名下	允许
http://www.a.com/lab/a.js http://www.a.com/script/b.js	同一域名下不同文件夹	允许
http://www.a.com:8000/a.js http://www.a.com/b.js	同一域名，不同端口	不允许
http://www.a.com/a.js https://www.a.com/b.js	同一域名，不同协议	不允许
http://www.a.com/a.js http://70.32.92.74/b.js	域名和域名对应 ip	不允许
http://www.a.com/a.js http://script.a.com/b.js	主域相同，子域不同	不允许
http://www.a.com/a.js http://a.com/b.js	同一域名，不同二级域名（同上）	不允许（cookie 这种情况下也不允许访问）
http://www.cnblogs.com/a.js	不同域名	不允许

http://www.a.com/b.js

特别注意两点：

第一，如果是协议和端口造成的跨域问题“前台”是无能为力的，

第二：在跨域问题上，域仅仅是通过“URL 的首部”来识别而不会去尝试判断相同的 ip 地址对应着两个域或两个域是否在同一个 ip 上。

“URL 的首部”指 window.location.protocol +window.location.host，也可以理解为“Domains, protocols and ports must match”。

接下来简单地总结一下在“前台”一般处理跨域的办法，后台 proxy 这种方案牵涉到后台配置，这里就不阐述了，有兴趣的可以看看 yahoo 的这篇文章：

《JavaScript: Use a Web Proxy for Cross-Domain XMLHttpRequest Calls》

1、document.domain+iframe 的设置

对于主域相同而子域不同的例子，可以通过设置 document.domain 的办法来解决。 具体的做法是可以在 http://www.a.com/a.html 和 http://script.a.com/b.html 两个文件中分别加上 document.domain = ‘a.com’；然后通过 a.html 文件中创建一个 iframe，去控制 iframe 的 contentDocument，这样两个 js 文件之间就可以“交互”了。当然这种办法只能解决主域相同而二级域名不同的情况，如果你异想天开的把 script.a.com 的 domian 设为 alibaba.com 那显然是会报错地！代码如下：

www.a.com 上的 a.html

```
1    document.domain = 'a.com';
2    var ifr = document.createElement('iframe');
3    ifr.src = 'http://script.a.com/b.html';
4    ifr.style.display = 'none';
5    document.body.appendChild(ifr);
6    ifr.onload = function(){
7            var doc = ifr.contentDocument || ifr.contentWindow.document;
8            // 在这里操纵 b.html
9            alert(doc.getElementsByTagName("h1")[0].childNodes[0].nodeValue);
10   };
```

script.a.com 上的 b.html

```
1    document.domain = 'a.com';
```

这种方式适用于{www.kuqin.com, kuqin.com, script.kuqin.com, css.kuqin.com}中的任何页面相互通信。

备注：某一页面的 domain 默认等于 window.location.hostname。主域名是不带 www 的域名，例如 a.com，主域名前面带前缀的通常都为二级域名或多级域名，例如 www.a.com 其实是二级域名。 domain 只能设置为主域名，不可以在 b.a.com 中将 domain 设置为 c.a.com。

问题：

1、安全性，当一个站点（b.a.com）被攻击后，另一个站点（c.a.com）会引起安全漏洞。

2、如果一个页面中引入多个 iframe，要想能够操作所有 iframe，必须都得设置相同 domain。

2、动态创建 script

虽然浏览器默认禁止了跨域访问，但并不禁止在页面中引用其他域的 JS 文件，并可以自由执行引入的 JS 文件中的 function（包括操作 cookie、Dom 等等）。根据这一点，可以方便地通过创建 script 节点的方法来实现完全跨域的通信。具体的做法可以参考 YUI 的 Get Utility
这里判断 script 节点加载完毕还是蛮有意思的：ie 只能通过 script 的 readystatechange 属性，其它浏览器是 script 的 load 事件。以下是部分判断 script 加载完毕的方法。

```
1  js.onload = js.onreadystatechange = function() {
2          if (!this.readyState || this.readyState === 'loaded' || this.readyState
   'complete') {
3                  // callback 在此处执行
4                  js.onload = js.onreadystatechange = null;
5          }
6  };
```

3、利用 iframe 和 location.hash

这个办法比较绕，但是可以解决完全跨域情况下的脚步置换问题。原理是利用 location.hash 来进行传值。在 url： http://a.com#helloword 中的'#helloworld' 就是 location.hash，改变 hash 并不会导致页面刷新，所以可 以利用 hash 值来进行数据传递，当然数据容量是有限的。假设域名 a.com 下的文件 cs1.html 要和 cnblogs.com 域名下的 cs2.html 传递信息，cs1.html 首先创建自动创建一个隐藏的 iframe，iframe 的 src 指向 cnblogs.com 域名下的 cs2.html 页面，这时的 hash 值可以做参数传递用。cs2.html 响应请求后再将通过修改 cs1.html 的 hash 值来传递数据（由于两个页面不在同一个域下 IE、Chrome 不允许修改 parent.location.hash 的值，所以要借助于 a.com 域名下的一个代理 iframe；Firefox 可以修改）。同时在 cs1.html 上加一个定时器，隔一段时间来判断 location.hash 的值有没有变化，一点有变化则获取获取 hash 值。代码如下：

先是 a.com 下的文件 cs1.html 文件：

```
function startRequest(){
        var ifr = document.createElement('iframe');
        ifr.style.display = 'none';
        ifr.src = 'http://www.cnblogs.com/lab/cscript/cs2.html#paramdo';
        document.body.appendChild(ifr);
}

function checkHash() {
        try {
                var data = location.hash ? location.hash.substring(1) : '';
                if (console.log) {
                        console.log('Now the data is '+data);
                }
        } catch(e) {};
}
setInterval(checkHash, 2000);
```

cnblogs.com 域名下的 cs2.html:

```
//模拟一个简单的参数处理操作
switch(location.hash){
        case '#paramdo':
                callBack();
                break;
        case '#paramset':
                //do something……
                break;
}

function callBack(){
        try {
                parent.location.hash = 'somedata';
        } catch (e) {
                // ie、chrome 的安全机制无法修改 parent.location.hash，
                // 所以要利用一个中间的 cnblogs 域下的代理 iframe
                var ifrproxy = document.createElement('iframe');
                ifrproxy.style.display = 'none';
                ifrproxy.src  = 'http://a.com/test/cscript/cs3.html#somedata';
注意该文件在"a.com"域下
                document.body.appendChild(ifrproxy);
        }
```

```
    }
```

a.com 下的域名 cs3.html

```
1   //因为 parent.parent 和自身属于同一个域，所以可以改变其 location.hash 的值
2   parent.parent.location.hash = self.location.hash.substring(1);
```

当然这样做也存在很多缺点，诸如数据直接暴露在了 url 中，数据容量和类型都有限等……

4、window.name 实现的跨域数据传输

文章较长列在此处不便于阅读，详细请看 window.name 实现的跨域数据传输。

5、使用 HTML5 postMessage

HTML5 中最酷的新功能之一就是 跨文档消息传输 Cross Document Messaging。 下一代浏览器都将支持这个功能：Chrome 2.0+、Internet Explorer 8.0+, Firefox 3.0+, Opera 9.6+, 和 Safari 4.0+ 。 Facebook 已经使用了这个功能，用 postMessage 支持基于 web 的实时消息传递。

otherWindow.postMessage(message, targetOrigin);

otherWindow: 对接收信息页面的 window 的引用。可以是页面中 iframe 的 contentWindow 属性；window.open 的返回值；通过 name 或下标从 window.frames 取到的值。

message: 所要发送的数据，string 类型。

targetOrigin: 用于限制 otherWindow，“*”表示不作限制

a.com/index.html 中的代码：

```
 <iframe id="ifr" src="b.com/index.html"></iframe>
1<script type="text/javascript">
2window.onload = function() {
3         var ifr = document.getElementById('ifr');
4         var targetOrigin = 'http://b.com';    // 若写成'http://b.com/c/proxy.html'
5效果一样
6                                                                                    //
7若写成'http://c.com'就不会执行 postMessage 了
8         ifr.contentWindow.postMessage('I was there!', targetOrigin);
9};
 </script>
```

b.com/index.html 中的代码：

```
<script type="text/javascript">
        window.addEventListener('message', function(event){
                // 通过 origin 属性判断消息来源地址
                if (event.origin == 'http://a.com') {
                        alert(event.data);        // 弹出"I was there!"
                        alert(event.source);    // 对 a.com 、 index.html 中
window 对象的引用
                                                                        // 但由
于同源策略，这里 event.source 不可以访问 window 对象
                }
        }, false);
</script>
```

参考文章：《精通 HTML5 编程》第五章——跨文档消息机制、
https://developer.mozilla.org/en/dom/window.postmessage

6、利用 flash

这是从 YUI3 的 IO 组件中看到的办法，具体可见
http://developer.yahoo.com/yui/3/io/。
可以看在 Adobe Developer Connection 看到更多的跨域代理文件规范：
ross-Domain Policy File Specifications、HTTP Headers Blacklist。
原文 http://blog.jobbole.com/53487/?utm_source=tuicool

JavaScript 社区开发者调查：服务端 JS 盛行，Backbone.js 使用最多

摘要：DailyJS 社区近日发起了一项针对 JavaScript 开发者的问卷调查，共有 3179 位开发者参与，结果显示，大部分开发者编写浏览器端 JavaScript，Backbone.js 框架使用最多，大部分开发者在 Github 中托管项目以及查找其他项目。

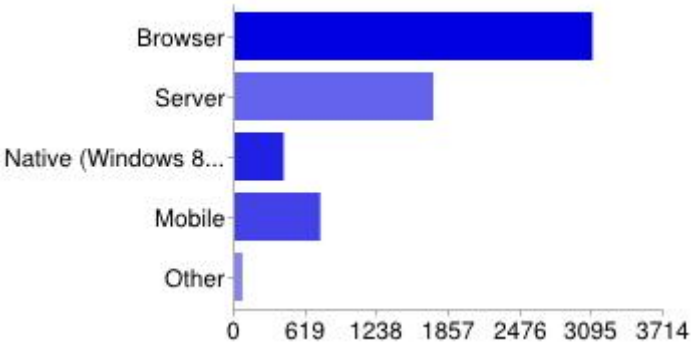
近日 DailyJS 社区发起了一项针对 JavaScript 开发者的问卷调查，共有 3179 位开发者参与回复，目前结果就已经出炉。

这些受访者中，其中 13%的开发者拥有 10 多年的 JavaScript 编写经验，24%的开发者拥有 5~10 年的经验，3~5 年经验的开发者最多，占 34%。下面就来看看这些开发者是如何使用 JavaScript 的。

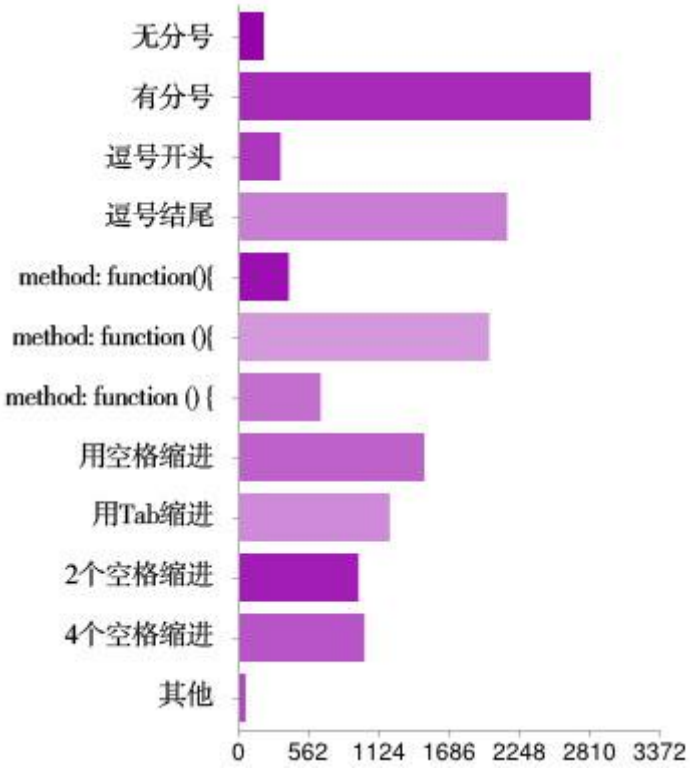
1. 你写什么类型的 JavaScript？

在调查中，大部分开发者都写过多种类型的 JavaScript。随着 Node.js 的盛行，越来越多的开发者开始写服务器端 JavaScript，此次调查中共有 1719 位写过服务器端 JavaScript。几乎所有开发者都写过浏览器端 JavaScript。

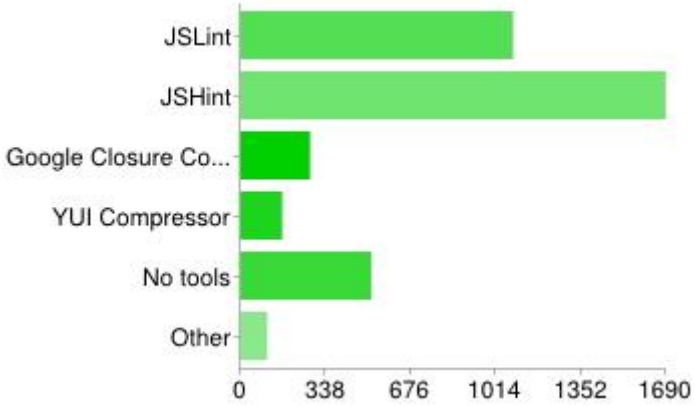
由于 JavaScript 跨平台的特性，其中一部分开发者也编写过本地（Windows 8、PhoneGap 等）和移动端的代码。



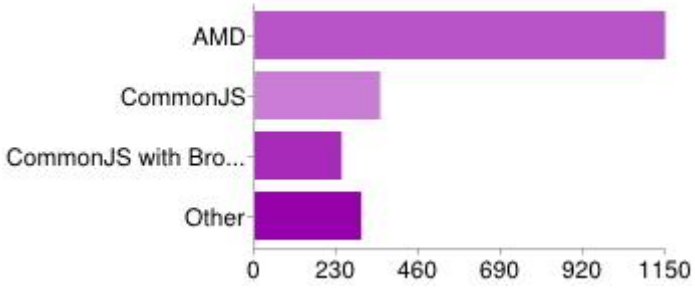
2. **你喜欢什么样的 JavaScript 代码风格？**



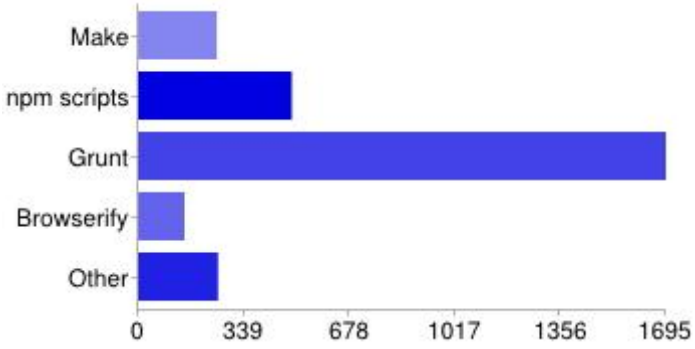
3. **你使用什么工具来验证代码质量？**



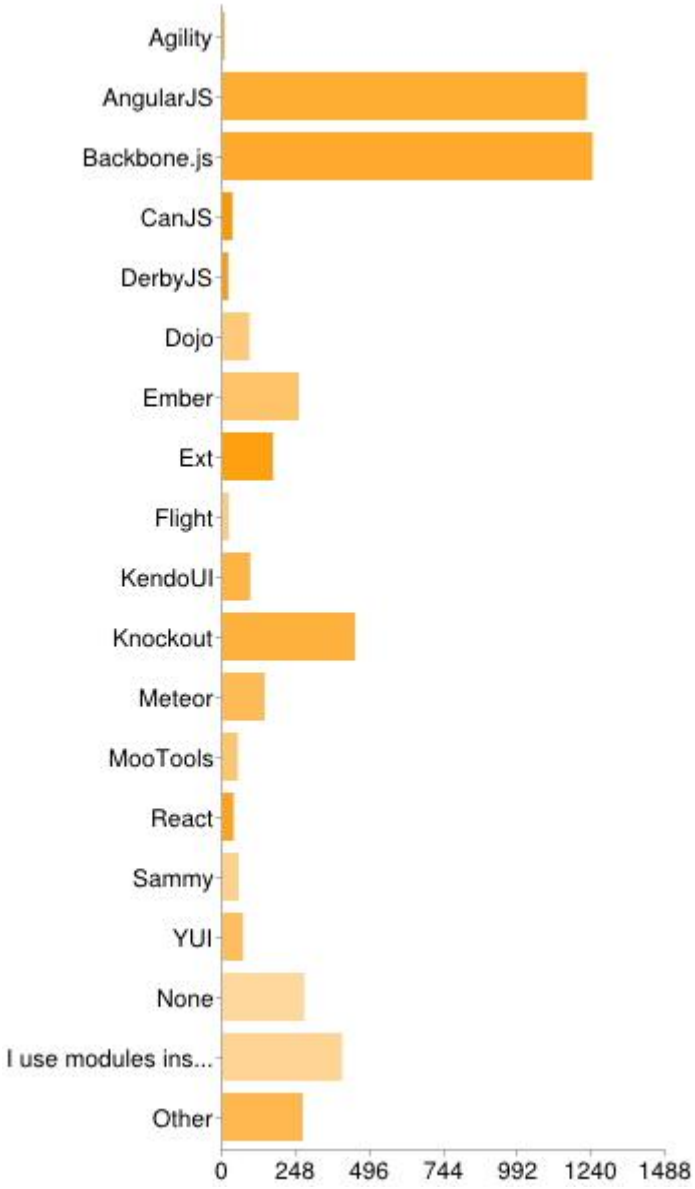
4. 你如何处理客户端依赖？



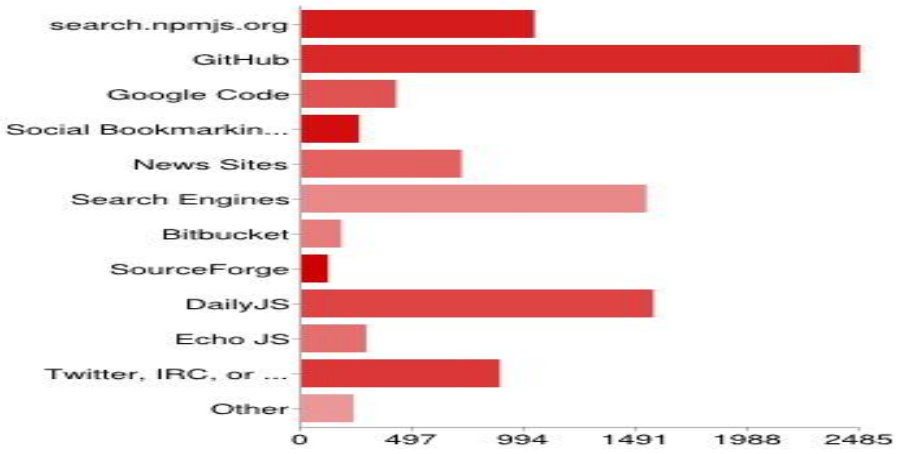
5. 你更喜欢用哪种脚本构建方案？



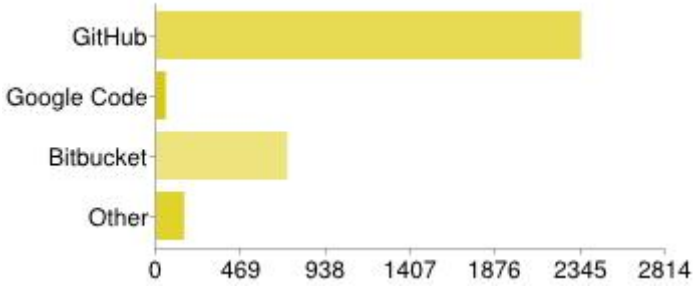
6. 你使用什么框架？



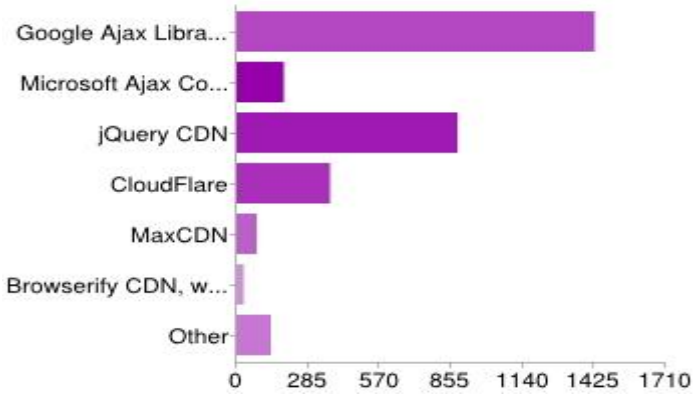
7. 你如何找到可重用的代码、库或工具？



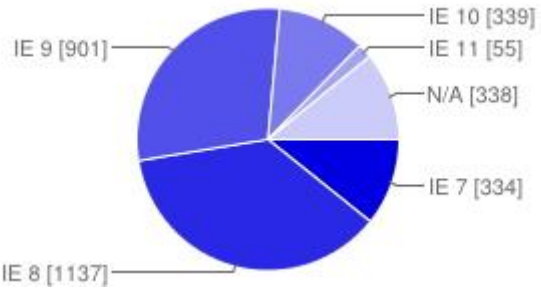
8. 你喜欢在什么地方托管你的 JavaScript 项目？



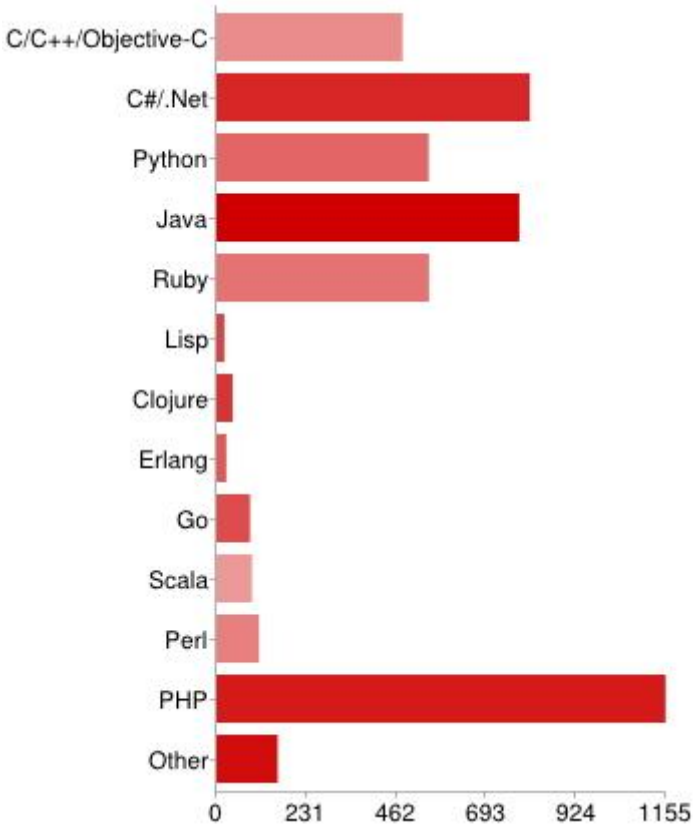
9. 你通过哪个 CND 服务来使用第三方库？



10. 你会为最低哪个 IE 版本进行兼容测试？



11. 除了 JavaScript 外，你的主要开发语言是什么？



12. 其他

- **JavaScript 使用场景**:2946 位开发者主要在工作中使用 JavaScript,2433 位开发者在业余项目中使用（其中部分开发者在这两种项目中都用到了 JavaScript）
- **所用语言**：78%的开发者直接使用 JavaScript 语言，22%的开发者使用其他语言，然后编译为 JavaScript。在使用其他语言的开发者中，CoffeeScript 使用得最多，占 64%，其次是 TypeScript，占 19%。
- **测试**：25%的开发者写测试，26%的开发者不写，其余的开发者有时候写，或在需要的时候写。在所用测试库中，Jasmine 使用最多，占 30%，其次是 Mocha，占 27%。
- **集成测试**:36%的开发者使用持续集成（CI）系统来运行测试，其中 Jenkins 使用最为广泛。
- **线下学习 JavaScript 的方式**：51%的开发者选择参加大会，49%的开发者选择参加小型沙龙。
- **ES6 特性**：15%的开发者已经在代码中使用了 ES6 的特性。

原文

http://www.csdn.net/article/2013-12-16/2817820-javascript-survey-results?utm_source=tuicool

CSS2.0 实现面包屑

面包屑这样的 我们以前都是用背景图片做这块工作，但是直到大概 2 个星期之前在新浪微博上看到用 css3.0 实现这样的面包屑 但是目前情况下 IE6-8 并不支持 css3.0 只有标准游览器(像火狐 谷歌等支持)。由于有前一次总结一篇关于"CSS 实现气泡框效果" 的文章 其中有关于怎么样实现小三角形的列子 所以感觉用那个小三角形可以正好模拟这块工作，所以也就试着做了一个。下面我们来看看面包屑大概是个什么样的效果！如下图：

如上所示：

思路：

1. 页面有 3 个 li li 标签嵌套有 2 个标签 分别模拟 2 个小三角形 第一个就是白色背景那块 第二块就是和灰色背景重叠的那个小三角。

HTML 代码可以写成如下：

```
<div class="box">
       <ul>
          <li>gggg<em></em><i></i></li>
          <li class="current">gggg<em></em><i></i></li>
          <li>gggg<em></em><i></i></li>
       </ul>
    </div>
```

下面我们这个实例效果先放放 我们还是来复习下以前写的 "css 实现气泡框效果" 中怎么样实现一个小三角形吧！

比如页面有如下 HTML 代码：

```
<div class="demo"></div>
```

现在我想用 css 实现一个小三角形 我们现在该如何做？先不急 慢慢来 一步一步拆分。

1. 首先我们来看看 css border 属性，当我们把一个 div border-color 设置成不同颜色时候，可以看到四边都成了矩形了。如下 css 代码

```
.demo {width:50px;height:50px;border-width:50px;border-style:solid;bo
rder-color:#CCC #00F #933 #0C9;}
```

如下图所示：

2. 如果我们继续把 div 的宽度和高度设为 0 的话 那么四边会成了三角形了。

如下图所示：

但是 IE6 下 上下是三角形 左右是矩形框：如下：

通过实验发现当把 div 的 font-size 和 line-height 都设为 0 的时候，div 的四边在 IE6 下都能形成完美的三角形：代码如下

```
.demo{width:0; height:0; border-width:50px; border-style:solid; border-color:#CCC #00F #933 #0C9;line-height:0;font-size:0;}
```

3. 很明白我们只需要一个三角形，那么我们只需要把其他三边颜色设置为透明或者设置为和背景颜色相同就可以制作出一个三角形出来了，将其他三边颜色设置为透明，即 color 的值为 transparent，如果其他三边颜色跟页面背景一样，虽然视觉上只能看到一个三角，但背景颜色一旦改变，其他三边颜色也要随之改变。如下代码：

```
.demo{width:0; height:0; border-width:50px; border-style:solid; border-color:#CCC transparent transparent transparent;line-height:0;font-size:0;}
```

但是在 IE6 下 又有问题了 IE6 不支持透明 transparent 如下：

但通过实验发现把 border-style 设置为 dashed 后，IE6 下其他三边就能透明了！如下：

现在小三角已经制作完毕！

现在面包屑的小三角该怎么做？

1. 首先我们看看 HTML 结构如下：

```
<div class="box">
       <ul>
          <li>gggg<em></em><i></i></li>
          <li class="current">gggg<em></em><i></i></li>
          <li>gggg<em></em><i></i></li>
       </ul>
    </div>
```

那么正常的情况下 我们添加如下 css 样式

```
*{ margin:0; padding:0;}
ul,li{list-style:none;}
.box{position:relative;margin:100px auto;background:#ccc;width:600px;
height:32px;line-height:32px;overflow:hidden;}
.box li{float:left;width:200px;text-align:center;position:relative;z-
index:2;}
```

可以实现如下效果：

gggg gggg gggg

2. 我们现在的问题是 我希望在每列右侧添加一个小三角形 背景为白色 覆盖到灰色背景上面去 所以我们可以在 em 标签上写 css 样式 制作小三角如上有怎么制作的 所以这里就不多说了. 代码如下：

```
.box li em{width:0;height:0;border-color:transparent transparent transparent #fff;border-style:dashed dashed dashed solid;border-width:24px 0 24px 24px;position:absolute;right:-24px;top:-8px;line-height:0;font-size:0;}
```

加上 css 代码后 效果图如下：

安照正常情况下 因为我是在每列右侧加一个小三角 且用了 overflow：hidden 所以最后一个小三角没有了，但是在 IE6，7 下 最后一个也有小三角 所以我在最外层容器

.box{position:relative};加了一个相对定位 所以目前兼容 IE6+ 火狐 谷歌等游览器。

3. 现在已经做成如上所示的样子 我们离我们想要的效果还没有完成 所以我们现在还需要在 li 标签上 需要再做个小三角形 背景为灰色 覆盖到上面去 CSS 代码如下：

```
.box li i{width:0;height:0;border-color:transparent transparent transparent #ccc;border-style:dashed dashed dashed solid;border-width:16px 0 16px 16px;position:absolute;right:-16px;top:0;line-height:0;font-size:0;}
```

4. 但是由于当前有选中的状态 所以还要把 current 样式加上 如下：

```
.box li.current{background:#933;z-index:1;}
.box li.current i{border-color:transparent transparent transparent #933;}
```

现在一切都完成了 效果如下：

现在把所有代码综合下：

HTML 代码如下：

```
<div class="box">
        <ul>
            <li>gggg<em></em><i></i></li>
            <li class="current">gggg<em></em><i></i></li>
            <li>gggg<em></em><i></i></li>
        </ul>
```

```
</div>
```

CSS 代码如下：

```
*{ margin:0; padding:0;}
ul,li{list-style:none;}
.box{position:relative;margin:100px auto;background:#ccc;width:600px;height:32px;line-height:32px;overflow:hidden;}
.box li{float:left;width:200px;text-align:center;position:relative;z-index:2;}
.box li em{width:0;height:0;border-color:transparent transparent transparent #fff;border-style:dashed dashed dashed solid;border-width:24px 0 24px 24px;position:absolute;right:-24px;top:-8px;line-height:0;font-size:0;}
.box li i{width:0;height:0;border-color:transparent transparent transparent #ccc;border-style:dashed dashed dashed solid;border-width:16px 0 16px 16px;position:absolute;right:-16px;top:0;line-height:0;font-size:0;}
.box li.current{background:#933;z-index:1;}
.box li.current i{border-color:transparent transparent transparent #933;}
```

原文

http://www.cnblogs.com/tugenhua0707/p/3485384.html?utm_source=tuicool

Grunt-前端利器

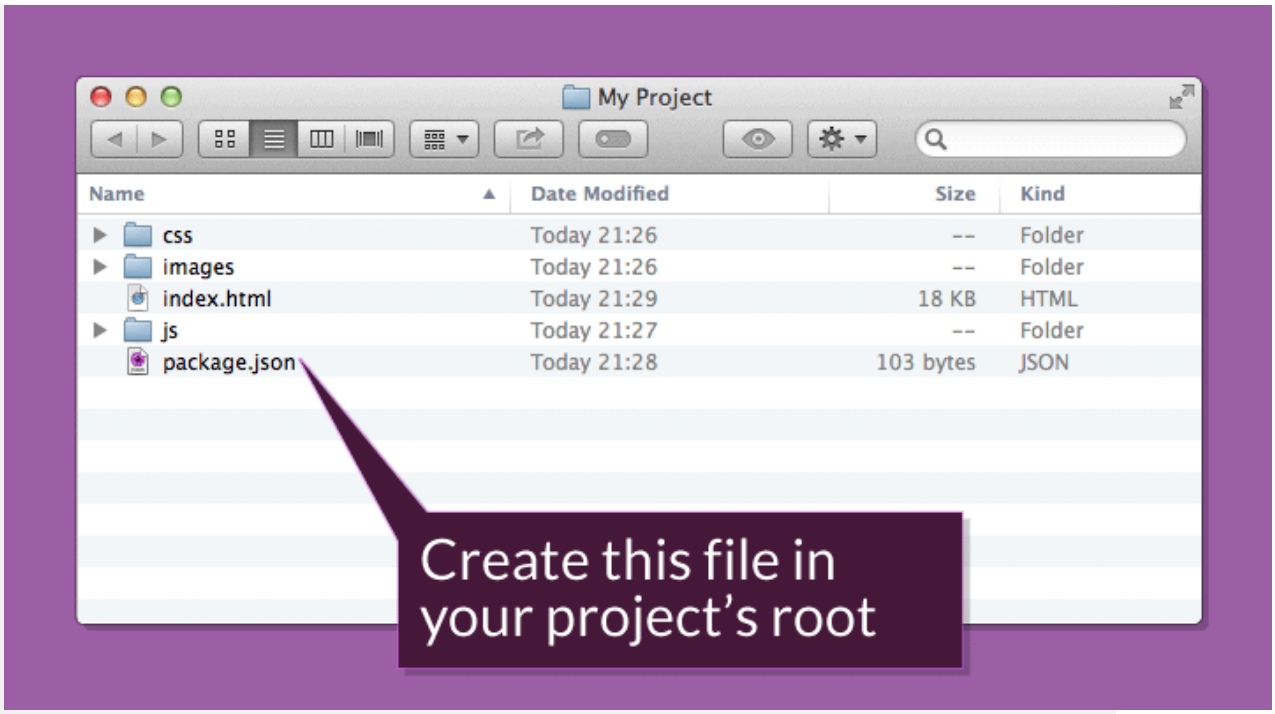
原文：Grunt for People Who Think Things Like Grunt are Weird and Hard
此篇笔记包含了个人的理解，只记录了文章的要点，并没有逐句翻译，如有不妥望指正。

Grunt 可以帮前端工程师做什么：

- 合并 CSS 和 JS 文件
- 压缩 CSS ， 最小化 JS
- 优化图片
- 使用 Sass

前提：安装 Node.js

项目根目录需要 **package.json** 文件



package.json 内容：

```
{
  "name": "example-project",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.1"
  }
}
```

打开命令行工具，切换到项目文件夹，运行命令：

```
npm install
```

node_modules 文件夹出现了，里面是 “**devDependencies**”属性定义的依赖包

全局安装 Grunt CLI：

```
npm install -g grunt-cli
```

合并文件

安装合并文件插件 grunt-contrib-concat

```
npm install grunt-contrib-concat --save-dev
```

package.json 会为” devDependencies“属性自动添加新的依赖，多了这一行：

```
"grunt-contrib-concat": "~0.3.0"
```

配置 **Gruntfile.js** 配置文件

格式如下：

```
module.exports = function(grunt) {

    // 1. 所有配置文件
    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json'),

        concat: {
            // 2.  合并文件的配置项
        }

    });

    // 3. 加载使用的插件
    grunt.loadNpmTasks('grunt-contrib-concat');

    // 4. 默认任务
    grunt.registerTask('default', ['concat']);

};
```

合并文件的插件配置实例：

```
concat: {
    dist: {
        src: [
            'js/libs/*.js', //libs 文件夹的所有 JS 文件
            'js/global.js'  // 指定文件
        ],
        dest: 'js/build/production.js',
    }
}
```

src 属性是原 JS 文件的数组，dest 是合并后的文件

比较全面 Gruntfile.js 文件看这个模版

然后在命令行运行：

```
grunt
```

最小化 JS

为 Grunt 添加新任务比较简单：

1. 找到需要的 Grunt 插件
2. 学习插件的配置风格
3. 为自己的项目写配置文件

官方的最小化插件 grunt-contrib-uglify，安装一下：

```
npm install grunt-contrib-uglify --save-dev
```

修改 Gruntfile.js 文件，加载此插件：

```
grunt.loadNpmTasks('grunt-contrib-uglify');
```

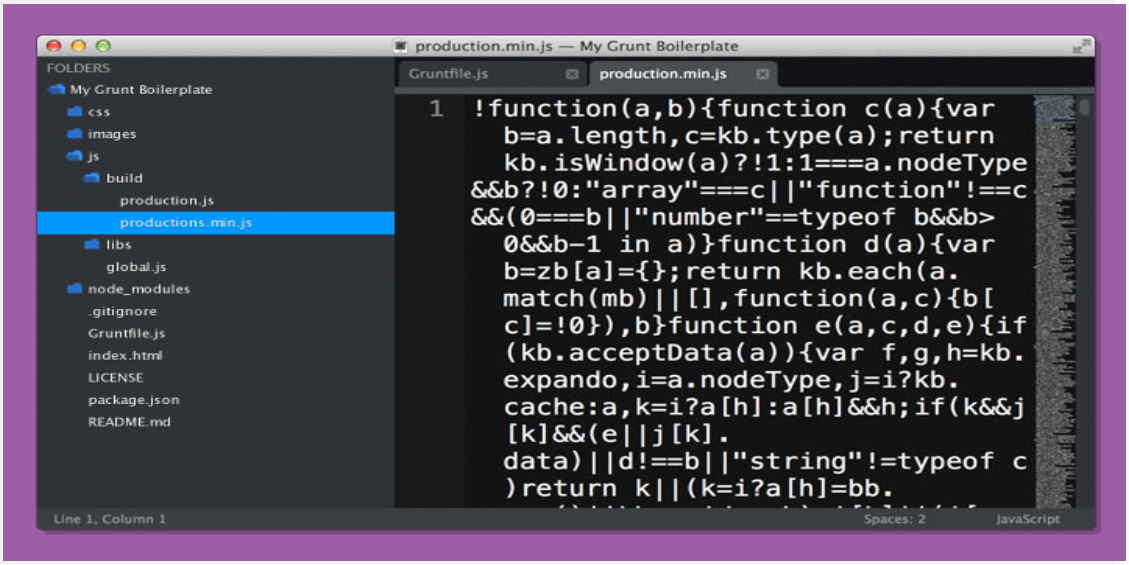
配置一下：

```
uglify: {
    build: {
        src: 'js/build/production.js',
        dest: 'js/build/production.min.js'
    }
}
```

更新默认任务：

```
grunt.registerTask('default', ['concat', 'uglify']);
```

运行 `grunt` 即可执行最小化任务



优化图片

官方的图片优化插件 grunt-contrib-imagemin
安装即可：

```
npm install grunt-contrib-imagemin --save-dev
```

修改 Gruntfile.js 文件，加载此插件：

```
grunt.loadNpmTasks('grunt-contrib-imagemin');
```

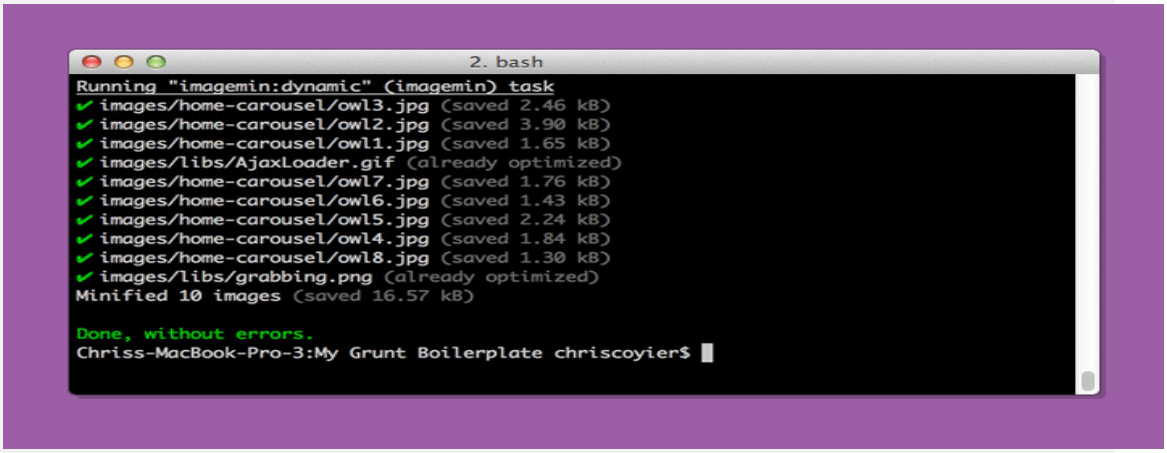
添加配置：

```
imagemin: {
    dynamic: {
        files: [{
            expand: true,
            cwd: 'images/',
            src: ['**/*.{png,jpg,gif}'],
            dest: 'images/build/'
        }]
    }
}
```

添加成默认任务

```
grunt.registerTask('default', ['concat', 'uglify', 'imagemin']);
```

运行 `grunt` 看奇迹发生



让任务更智能，更自动

1. 需要执行的时候自动执行
2. 一次执行一个任务

比如:

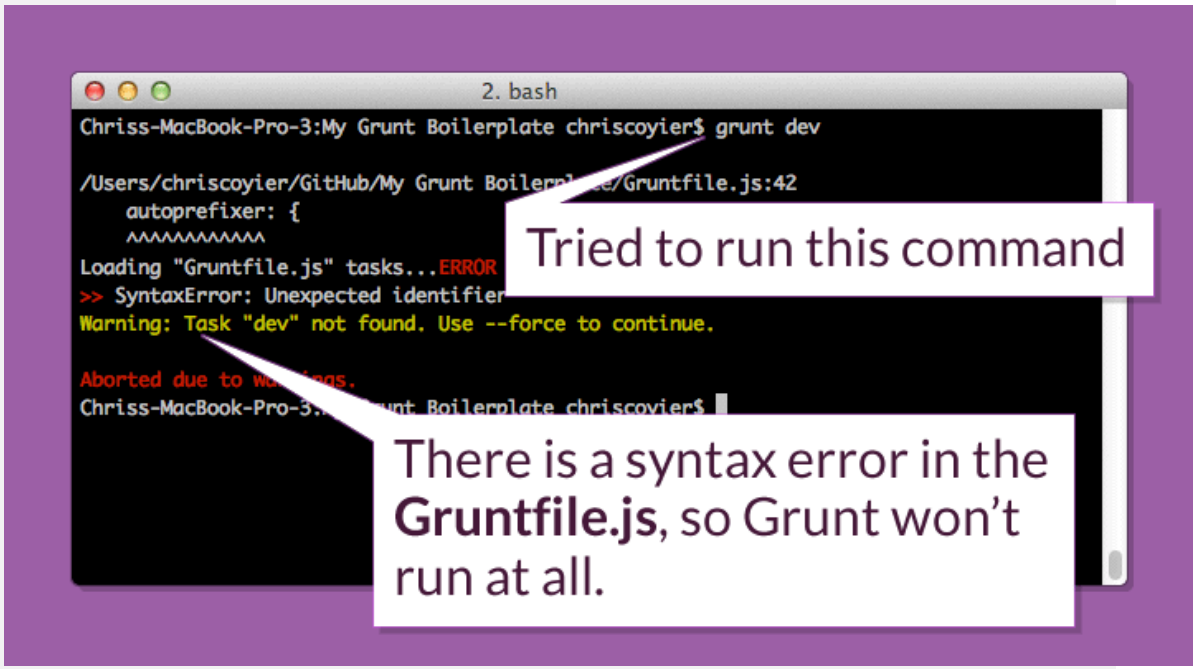
1. 当 JS 修改了以后，合并，最小化文件
2. 当添加新图片，修改旧图片时优化图像

可以通过官方的 grunt-contrib-watch 插件监视文件。

照例安装插件，修改配置文件：

```
watch: {
    scripts: {
        files: ['js/*.js'],
        tasks: ['concat', 'uglify'],
        options: {
            spawn: false,
        },
    }
}
```

出现错误的时候，命令行会有提示：



Grunt 做我们的预处理器

我还没用过此类的预处理器，好 out .....先留个坑，等用的时候再做记录。请感兴趣的同学参考原文

继续升级

添加更多有用的任务：

- 使用 Autoprefixer，CSS 中自动加前缀
- JS 单元测试（比如：Jasmine ）
- 自动生成图片雪碧图和 SVG 图标（比如：Grunticon ）
- 作为文件资源服务器
- 代码质量工具： HTML-Inspector， CSS Lint 或者 JS Hint
- 跨浏览器 CSS 注入
- 帮助提交版本控制仓库，比如 GitHub
- 添加资源版本号（清缓存）
- 帮助部署测试和生成环境（比如：DPLOY ）

原文

http://jinlong.github.io/blog/2013/12/22/grunt-for-fe/?utm_source=tuicool

跨浏览器开发工作小结

本篇小结是在 2011 年时候总结的，当时做一个产品的跨浏览器兼容工作，由于产品开发的时间比较早，最开始只能在 IE 下面(IE 8、IE 9 还有点点问题)使用，做跨浏览器兼容工作的时候，主要是适配 IE 6--IE 9、Safari、FireFox、Chrome，引入了 jQuery 框架进行改造后，大部分功能可以正常使用，现将总结分享一下。

1.eval(idName)

【问题描述】：IE、safari、Chrome 浏览器下都可以使用 eval(idName)或 getElementById(idName)来取得 id 为 idName 的 HTML 对象；firefox 下只能使用 getElementById(idName)来取得 id 为 idName 的 HTML 对象.

【兼容办法】：统一用 getElementById("idName")来取得 id 为 idName 的 HTML 对象。

2.ActiveXObject

【问题描述】：IE 下支持用 var obj = new ActiveXObject() 的方式创建对象，但其它浏览器都会提示 ActiveXObject 对象未定义。

【兼容办法】：

(1)在使用 new ActiveXObject()之前先判断浏览器是否支持 ActiveXObject 对象，以创建 AJAX 对象为例：

```
if(window.ActiveXObject)
{
    this.req=new ActiveXObject("Microsoft.XMLHTTP");
}
else if(window.XMLHttpRequest)
{
    this.req=new XMLHttpRequest();
}
```

(2)使用 jQuery 封装的 ajax 方法来创建对象，以创建 AJAX 对象为例(推荐):

```
var strResponse = "";
jQuery.ajax({ url: sAspFile, data: "<root>" + sSend + "</root>", processData: false, async: false, type: "POST",
    error: function(XMLHttpRequest, textStatus, errorThrown)
    {
            strResponse = textStatus;
    },
     success: function(data, textStatus)
     {
            strResponse = data;
    }
});
```

3.XML 操作

【问题描述】：通常装载 xml 文档使用 ActiveXObject 对象，但除非 IE 外，其它浏览器都不支持此方法。XML 文档操作，IE 和其它浏览器也存在不同，通常取 XML 对象的 XML 文本的方法是 xml.documentElement.xml，但 xml 属性只有 IE 支持，其它浏览器均不支持。查找节点是常用的方法有 selectNodes 和 selectSingleNode，这两个方法也只有 IE 支持，其它浏览器需要自己扩展。

【兼容办法】

(1)装载 XML 文档：用$.ajax()，参考 jquery 帮助文档

(2)xml 对象转字符串，如：

```
var stringtoxml = function(str) { //字符串转 xml 对象
    var s = "<?xml version='1.0' encoding='utf-8' ?>" + str;
    var objxml = null;
    if (window.ActiveXObject) {
        objxml = new ActiveXObject("Microsoft.XMLDOM");
        objxml.async = false;
        objxml.loadXML(s);
    }
    else {
```

```
        objxml = (new DOMParser()).parseFromString(s, "text/xml");
    }
    return objxml;
}

var xmltostring = function(dom) {  //xml 对象转字符串
    if (dom instanceof jQuery) {
        dom = dom[0];
    }
    var str = null;
    if (window.ActiveXObject) {
        str = dom.xml;
    }
    else {
        str = (new XMLSerializer()).serializeToString(dom);
    }
    return str;
}

var oXML0 = stringtoxml("<root>"+ xml +"</root>");
var root = oXML0.documentElement;
var strXml = xmltostring(root).replace("<root>","");
```

(3)字符串转 xml 对象，如：

```
var oXML = stringtoxml("<root>" +
document.getElementById("hidTaskXml").value + "</root>");
```

(4)查找结点：可以用 JQUERY 同的 find 方法来查找结点，如：

```
var item = $(oXML).find("record");
```

或者用原型扩展方法为 XML 对象添加 selectNodes 和 selectSingleNode 方法，扩展方法如下：

```
if( document.implementation.hasFeature("XPath", "3.0") )
{
    XMLDocument.prototype.selectNodes =function(cXPathString, xNode)
    {
        if( !xNode )
        {
            xNode = this;
        }
        var oNSResolver = this.createNSResolver(this.documentElement);

        var aItems = this.evaluate(cXPathString, xNode, oNSResolver,

        XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null) ;
```

```
        var aResult = [];

        for( var i = 0; i < aItems.snapshotLength; i++)
        {
            aResult[i] = aItems.snapshotItem(i);
        }

        return aResult;
    }

    Element.prototype.selectNodes = function(cXPathString)
    {
        if(this.ownerDocument.selectNodes)
        {
            return this.ownerDocument.selectNodes(cXPathString, this);
        }
        else
        {
            throw "For XML Elements Only";
        }
    }

    XMLDocument.prototype.selectSingleNode = function(cXPathString,
xNode)
    {
        if( !xNode )
        {
           xNode = this;
        }
        var xItems = this.selectNodes(cXPathString, xNode);
        if( xItems.length > 0 )
        {
             return xItems[0];
        }
        else
        {
            return null;
        }
    }

    Element.prototype.selectSingleNode = function(cXPathString)
    {
        if(this.ownerDocument.selectSingleNode)
```

```
        {
            return this.ownerDocument.selectSingleNode(cXPathString,
this);
        }
        else
        {
            throw "For XML Elements Only";
        }
    }
}
```

4.window.execScript()

【问题描述】：只有 IE 浏览器支持 execScript 方法，其它的都不支持。但所有浏览器都支持 window.eval()方法。

【兼容办法】：用 window.eval()方法代替 window.execScript()。如

```
1 //window.execScript(“alert(123)”);
2
3 window.eval(“alert(123)”);
```

5.window.createPopup()

【问题描述】：创建一个弹出窗口的方法，IE 支持此方法，Safari、FireFox、Chrome 都不支持，使用时会提示 createPopup 方法未定义。

【兼容办法】：可用如下方法为 window 对象添加 createPopup 方法。

```
if (!window.createPopup) {
    var __createPopup = function() {
        var SetElementStyles = function( element, styleDict ) {
            var style = element.style ;
            for ( var styleName in styleDict )style[ styleName ] =
styleDict[ styleName ] ;
        }
        var eDiv = document.createElement( 'div' );
        SetElementStyles( eDiv, {'position':'absolute', 'top': 0 +'px',
'left': 0 +'px', 'width': 0 +'px', 'height': 0 +'px', 'zIndex': 1000,
'display' : 'none', 'overflow' : 'hidden' } ) ;
        eDiv.body = eDiv ;
        var opened = false ;
        var setOpened = function( b ) {
            opened = b;
        }

        var getOpened = function() {
            return opened ;
        }
```

```
        var getCoordinates = function( oElement ) {
            var coordinates = {x:0,y:0} ;
            while( oElement ) {
                coordinates.x += oElement.offsetLeft ;
                coordinates.y += oElement.offsetTop ;
                oElement = oElement.offsetParent ;
            }
            return coordinates ;
        }
        return {
            htmlTxt : '',
      document : eDiv,
      isOpen : getOpened(),
      isShow : false,
      hide : function() {
       SetElementStyles( eDiv, { 'top': 0 + 'px', 'left': 0 + 'px',
'width': 0 + 'px', 'height': 0 + 'px', 'display' : 'none' } ) ;
       eDiv.innerHTML = '' ;
       this.isShow = false ;
      },
      show : function( iX, iY, iWidth, iHeight, oElement ) {
       if (!getOpened()) {
        document.body.appendChild( eDiv ) ; setOpened( true ) ;
       } ;
       this.htmlTxt = eDiv.innerHTML ;
       if (this.isShow) {
           this.hide() ;
       } ;
       eDiv.innerHTML = this.htmlTxt ;
       var coordinates = getCoordinates ( oElement ) ;
       eDiv.style.top = ( iX + coordinates.x ) + 'px' ;
       eDiv.style.left = ( iY + coordinates.y ) + 'px' ;
       eDiv.style.width = iWidth + 'px' ;
       eDiv.style.height = iHeight + 'px' ;
       eDiv.style.display = 'block' ;
       this.isShow = true ;
      }
  }
    }
    window.createPopup = function() {
        return __createPopup();
    }
}
```

6.getYear()方法

【问题描述】：如下代码：

```
var year= new Date().getYear();

document.write(year);
```

在 IE 中得到的日期是"2011"，在 Firefox 中看到的日期是"111"，主要是因为在 Firefox 里面 getYear 返回的是"当前年份-1900" 的值。

【兼容办法】：解决办法是加上对年份的判断，如：

```
var year= new Date().getYear();
year = (year<1900?(1900+year):year);
document.write(year);
```

也可以通过 getFullYear getUTCFullYear 去调用：

```
var year = new Date().getFullYear();

document.write(year);
```

7.document.all

【问题描述】:document.all 在 IE、Safari 下都可以使用，firefox、Chrome 下不能使用

【兼容办法】：所有以 document.all.*方法获取对象的地方都改为 document.getElementById 或 document.getElementsByName 或 document.getElementsByTagName。

8.变量名与对象 ID 相同的问题

【问题描述】：IE 下,HTML 对象的 ID 可以作为 document 的下属对象变量名直接使用，如下面的写法：

```
objid.value = “123”;//objid 为控件 ID
```

其它浏览器下则不能这样写。原因是其它浏览器下,可以使用与 HTML 对象 ID 相同的变量名，IE 下则不能。

【兼容办法】：使用 document.getElementById(idName)等通用方法先获取对象，再操行其它操作。如：

```
document.getElementById(objid).value = “123”; //objid 为控件 ID
```

注：最好不要取 HTML 对象 ID 相同的变量名,以减少错误;在声明变量时,一律加上 var,以避免歧义。

9.select 元素的 add 方法

【问题描述】：在 IE，Safari，Chrome 下，select 控件添加项时使用如下的方法：

```
document.getElementById(“select1”).add(new Options(“aa”,” aa”));
```

但在 FireFox 下这样操作会报错。

【兼容办法】：统一使用兼容方法，加 options 属性，如下：

```
document.getElementById(“select1”).options.add(new
Options(“aa”,” aa”));
```

10.html 元素的自定义属性

【问题描述】:IE 下元素属性访问方法如 document.getElementById(id).属性名，而且对于自定义属性和非自定义属性均有效。但在其它浏览器下该方法只适应于元素的公共属性，自定义属性则取不到。

【兼容办法】：用 jQuery 的方法来取，如$("#id").attr("属性")或用 document.getElementById(id).getAttribute("属性")，两种方法都可以适用所有浏览器。

11.html 元素 innerText 属性

【问题描述】：取元素文本的属性 innerText 在 IE 中能正常工作，但此属性不是 DHTML 标准，其它浏览器不支持，其它浏览器中使用 textContent 属性获取。

【兼容办法】：

(1)通用方法是用 jQuery 方法$("#id").text()，如：

```
//document.getElementById(id).innerText;

$("#id").text();
```

(2)取值前判断浏览器，根据具体情况取值，如：

```
var obj = document.getElementById(id);

var str = (obj.innerText)?obj.innerText:obj.textContent;
```

(3)也可以通过原型扩展方法来为元素添加 innerText，扩展方法如下：

```
if(typeof(HTMLElement)!="undefined" && !window.opera)
{
    var pro = window.HTMLElement.prototype;

    pro.__defineGetter__("innerText",function (){
        var anyString = "";
        var childS = this.childNodes;
        for(var i=0; i<childS.length; i++)
        {
            if(childS[i].nodeType==1)
            {
                anyString += childS[i].tagName=="BR" ? '\n' :
childS[i].innerText;
            }
            else if(childS[i].nodeType==3)
            {
                anyString += childS[i].nodeValue;
            }
        }
        return anyString;
    });

    pro.__defineSetter__("innerText",function(sText){
        this.textContent=sText;
    });
}
```

12.html 元素 innerHTML、outerHTML 属性

【问题描述】：innerHTML 是所有浏览器都支持的属性。outerHTML 属性不是 DHTML 标准，IE 外的其它浏览器不支持。

【兼容办法】：在非 IE 浏览器下必须使用扩展方法才能获取，扩展方法如下：

```
if(typeof(HTMLElement)!="undefined" && !window.opera)
{
	var pro = window.HTMLElement.prototype;
	pro.__defineGetter__("outerHTML", function(){
		var str = "<" + this.tagName;
		var a = this.attributes;
		for(var i = 0, len = a.length; i < len; i++)
		{
			if(a[i].specified)
			{
				str += " " + a[i].name + '="' + a[i].value + '"';
			}
		}
		if(!this.canHaveChildren)
		{
			return str + " />";
		}
		return str + ">" + this.innerHTML + "</" + this.tagName + ">";
	});

	 pro.__defineSetter__("outerHTML", function(s){
		var r = this.ownerDocument.createRange();
		r.setStartBefore(this);
		var df = r.createContextualFragment(s);
		this.parentNode.replaceChild(df, this);
		return s;
	});
}
```

13.html 元素 parentElement 属性

【问题描述】：parentElement 是取元素父结点的属性，此属性只有 IE 支持，其它浏览器均不支持。

【兼容办法】：用 parentNode 属性来获取父结点，如：

```
//document.getElementById(id).parentElement;

document.getElementById(id).parentNode;
```

14.集合类对象问题

【问题描述】：IE 下对于集合类对象，如 forms,frames 等,可以使用()或[]获取集合类对象，Safari，Chrome 也都支持，如 document.forms(“formid”) 或 document.forms[“formid”]。但 Firefox 下,只能使用[]获取集合类对象。

【兼容办法】：统一使用[]获取集合类对象，如：

```
document.forms[0];

document.forms[“formid”];
```

【注】：所有以数组方式存储的对象都在访问子成员时，都必须以[]方式索引得到，如常见的 XML 文档遍历，也需要改，如下：

```
// xmldoc.documentElement.childNodes(1)

xmldoc.documentElement.childNodes[1]
```

15.frame 操作

【问题描述】:在 IE、Safari、Chrome 下，用 window 对象访问 frame 对象时，可以用 id 和 name 属性来获取，如

```
window.frameId;
window.frameName;
```

但在 firefox 下，必须使用 frame 对象的 name 属性才能获取到。

【兼容办法】：

(1)访问 frame 对象：统一使用 window.document.getElementById(frameId)来访问这个 frame 对象。

(2)切换 frame 内容：统一使用 window.document.getElementById(testFrame).src=xxx.htm 切换。

如果需要将 frame 中的参数传回父窗口，可以在 frame 中使用 parent 来访问父窗口。例如：parent.document.form1.filename.value=Aqing;

(3)iframe 页中的对象: $("#frameid").contents().find("#html 控件 id")

(4)iframe 页中的 iframe: $("#frameid").contents().find("#frameid1").contents();

(5)iframe 中的方法或变量: $("#frameid")[0] .contentWindow.SaveFile("false", strRet, a);

16.insertAdjacentHTML 和 insertAdjacentText

【问题描述】：insertAdjacentHTML 方法是比 innerHTML、outerHTML 属性更灵活的插入 HTML 代码的方法。它可以实现在一个 DOM 元素的前面、后面、第一个子元素前面、最后一个子元素后面四个位置，插入指定的 HTML 代码。不是 W3C 标准的 DOM 方法，W3C 近期在 HTML5 草案中扩展了这个方法。

insertAdjacentText 是比 innerText、outerText 属性更灵活的插入文本的方法。它可以实现在一个 DOM 元素的前面、后面、第一个子元素前面、最后一个子元素后面四个位置，插入指定的文本。不是 W3C 标准的 DOM 方法，至今为止 W3C 的 HTML5 还未涉及此方法。

insertAdjacentHTML 和 insertAdjacentText 可以 IE、Safari、Chrome 上执行，只有 FireFox 不支持，

【兼容办法】：可用以下方法进行扩展：

```
if (typeof(HTMLElement) != "undefined")
{
    HTMLElement.prototype.insertAdjacentElement = function(where,
parsedNode)
    {
        switch (where)
        {
            case "beforeBegin":
                this.parentNode.insertBefore(parsedNode, this);
                break;
            case "afterBegin":
                this.insertBefore(parsedNode, this.firstChild);
                break;
            case "beforeEnd":
                this.appendChild(parsedNode);
                break;
            case "afterEnd":
                if (this.nextSibling)
                    this.parentNode.insertBefore(parsedNode,
this.nextSibling);
                else
                    this.parentNode.appendChild(parsedNode);
                break;
        }
    }
    HTMLElement.prototype.insertAdjacentHTML = function(where,
htmlStr)
    {
        var r = this.ownerDocument.createRange();
        r.setStartBefore(this);
        var parsedHTML = r.createContextualFragment(htmlStr);
        this.insertAdjacentElement(where, parsedHTML);
    }

    HTMLElement.prototype.insertAdjacentText = function(where, txtStr)
    {
        var parsedText = document.createTextNode(txtStr);
        this.insertAdjacentElement(where, parsedText);
    }
}
```

17.Html 元素的 children 属性

【问题描述】：children 是取 HTML 元素子结点的属性，只有 IE 下支持，其它浏览器下用 childNodes。

【兼容办法】：统一改为用 childNodes 属性取子结点。或用以下方法扩展 HTML 元素的属性：

```
if (typeof(HTMLElement) != "undefined")
{
    HTMLElement.prototype.__defineGetter__("children",function(){
        var returnValue = new Object();
        var number = 0;
        for(var i=0; i<this.childNodes.length; i++)
        {
            if(this.childNodes[i].nodeType == 1)
            {
                returnValue[number] = this.childNodes[i];
                number++;
            }
        }
        returnValue.length = number;
        return returnValue;
    })
}
```

18.insertRow\inserCell

【问题描述】：insertRow 和 insertCell 是在表格中插入行或插入列的方法，在 IE 中使用方法如下

```
var nowTB = document.getElementById("tb1");
nowTR = nowTB.insertRow();
nowTD = nowTR.insertCell();
```

Safari、Chrome 下也可以正常执行，但插入行的位置不一样 IE 下默认在表尾插入行，Safari、Chrome 默认在表头插入行；但在 FireFox 下调用会报错。

【兼容办法】：下面的方法可以在所有浏览器上调用，而且插入行的位置都是表尾，不同之处就是执行前传递一个默认值。推荐使用。

```
var nowTB = document.getElementById("tb1");

nowTR = nowTB.insertRow(-1);

nowTD = nowTR.insertCell(-1);
```

19.document.createElement

【问题描述】：IE 有 3 种方式都可以创建一个元素：

```
1 document.createElement("<input type=text>")

2 document.createElement("<input>")
```

```
3 document.createElement("input")
```

Safari、FireFox、Chrome 只支持一种方式:

```
document.createElement("input");

document.setAttribute(name,value);
```

【兼容办法】：统一使用所有浏览器都支持的方法，如下：

```
document.createElement("input");

document.setAttribute(name,value);
```

20.浏览器处理 **childNodes** 的异同

【问题描述】：如下 HTML 代码：

```
<ul id="main">

<li>1</li>

<li>2</li>

<li>3</li>

</ul>

<input type=button value="click me!" onclick=

"alert(document.getElementById('main').childNodes.length)">
```

分别用 IE 和其它浏览器运行，IE 的结果是 3，而其它则是 7。

IE 是将一个完整标签作为一个节点，而 Safari、FireFox、Chrome 除了上述的的情况外，也把一个标签的结束符“>”到下一个标签的起始符“<”之间的内容（除注释外，包括任何的文字、空格、回车、制表符）也算是一个节点了，而且这种节点也有它们自己独特的属性和值 nodeName="#text"。

【兼容办法】：在实际运用中，Safari、FireFox、Chrome 在遍历子节点时，在 for 循环里加上

if(childNode.nodeName=="#text") continue;或者 nodeType == 1 这样，便跳过不需要的操作，使程序运行的更有效率。也可以用 node.getElementsByTagName()回避。

21.document.getElementsByName

【问题描述】：在元素只有 name 属性，没有 id 属性的情况下，在 IE 中获取不到 DIV 元素，其它浏览器可以获取。当前 name 和 id 属性都存在时，所有浏览器都可以获取到 DIV 元素。

【兼容办法】：尽量用 ID 来获取。

22.tr 操作

【问题描述】：IE 下 table 中无论是用 innerHTML 还是 appendChild 插入<tr>都没有效果，因为在 IE 浏览器下 tr 是只读的。而其他浏览器下可以这样操作。

【兼容办法】：将<tr>加到 table 的<tbody>元素中，如下面所示：

```
var row = document.createElement("tr");

var cell = document.createElement("td");

var cell_text = document.createTextNode("插入的内容");

cell.appendChild(cell_text);

row.appendChild(cell);

document.getElementsByTagName("tbody")[0].appendChild(row);
```

23.移除节点 removeNode()

【问题描述】：appendNode 在 IE 和其它浏览器下都能正常使用，但是 removeNode 只能在 IE 下用。removeNode 方法的功能是删除一个节点，语法为 node.removeNode（false）或者 node.removeNode（true），返回值是被删除的节点。

removeNode（false）表示仅仅删除指定节点，然后这个节点的原孩子节点提升为原双亲节点的孩子节点。

removeNode（true）表示删除指定节点及其所有下属节点。被删除的节点成为了孤立节点，不再具有有孩子节点和双亲节点。

【兼容办法】：兼容 IE 和其它浏览器的方法是 removeChild，先回到父节点，在从父节点上移除要移除的节点。

```
// 为了在 IE 和其它浏览器下都能正常使用，取上一层的父结点，然后 remove。
node.parentNode.removeChild(node);
```

24.expression

【问题描述】：IE 下样式支持计算表达式 expression，但其它浏览器不支持，而且 IE 以后高版本也可能不再支持这种样式，所以不允许使用。下面是通常使用的情况：

```
<div id=”content”
 style=’height:expression(document.body.offsetHeight-80)” ></div>
```

【兼容办法】：去掉样式设置，将其写到函数中，分别在页面加载完毕和页面尺寸发生变化时执行。如下：

```
$(function(){
  $(“#content”).height($(document.body).height()-80);
})

$(window).resize(function(){
```

```
  $(“#content”).height($(document.body).height()-80);
});
```

25.Cursor

【问题描述】：Cursor 的 hand 属性只有 IE 支持，其它浏览器没有效果，如：

```
<div style=” cursor:hand” ></div>
```

【兼容办法】：统一用 pointer 值，如：

```
<div style=” cursor: pointer” ></div>
```

26.CSS 透明问题

【问题描述】：IE 支持但其它浏览器不支持的透明样式如下：

```
<div
style="filter:progid:DXImageTransform.Microsoft.Alpha(style=0,opacity
=20);width:200px;height:200px;background-color:Blue">ddddd</div>
```

其它浏览器支持但 IE 不支持的透明样式如下：

```
<div
style="opacity:0.2;width:200px;height:200px;background-color:Blue">dd
ddd</div>
```

【兼容办法】：利用"!important"来设置元素的样式。Safari，FireFox，Chrome 对于"!important"会自动优先解析，然而 IE 则会忽略。如下

```
<div
style="filter:progid:DXImageTransform.Microsoft.Alpha(style=0,opacity
=20);width:200px;height:200px;background-color:Blue;!important;
opacity:0.2">ddddd</div>
```

27.pixelHeight\pixelWidth

【问题描述】：pixelHeight\pixelWidth 是元素的高度和宽度样式，通常获取方法是：

```
obj.style.pixelWidth;

obj.style.pixelHeight;
```

IE，Safari，Chrome 都支持此样式，返回的值是整数，FireFox 不支持

【兼容办法】：所有浏览器都支持 obj.style.height，但返回的值是带单位的，如“100px”。可以用如下方法来获取：

```
parseInt(obj.style.height)
```

28.noWrap

【问题描述】：nowrap 属性是被废弃的属性。

【兼容办法】：使用 CSS 规则 white-space:nowrap 代替这个属性。

29.CSS 的 **float** 属性

【问题描述】：Javascript 访问一个给定 CSS 值的最基本句法是：object.style.property，但部分 CSS 属性跟 Javascript 中的保留字命名相同，如"float"，"for"，"class"等，不同浏览器写法不同。

在 IE 中这样写：

```
document.getElementById("header").style.styleFloat = "left";
```

在其它浏览器中这样写：

```
document.getElementById("header").style.cssFloat = "left";
```

【兼容办法】：兼容方法是在写之前加一个判断，判断浏览器是否是 IE：

```
if(jQuery.browser.msie){
    document.getElementById("header").style.styleFloat = "left";
}
else{
    document.getElementById("header").style.cssFloat = "left";
}
```

30.访问 **label** 标签中的 **for**

【问题描述】：for 属性规定 label 与哪个表单元素绑定。在 IE 中这样写：

```
var myObject = document.getElementById("myLabel");

var myAttribute = myObject.getAttribute("htmlFor");
```

在 Firefox 中这样写：

```
var myObject = document.getElementById("myLabel");

var myAttribute = myObject.getAttribute("for");
```

【兼容办法】：判断浏览器是否是 IE：

```
var myObject = document.getElementById("myLabel");
if(jQuery.browser.msie){
    var myAttribute = myObject.getAttribute("htmlFor");
}
else{
    var myAttribute = myObject.getAttribute("for");
}
```

31.访问和设置 **class** 属性

【问题描述】：同样由于 class 是 Javascript 保留字的原因，这两种浏览器使用不同的 JavaScript 方法来获取这个属性。

IE8.0 之前的所有 IE 版本的写法：

```
var myObject = document.getElementById("header");

var myAttribute = myObject.getAttribute("className");
```

适用于 IE8.0 以及 firefox 的写法：

```
var myObject = document.getElementById("header");

var myAttribute = myObject.getAttribute("class");
```

另外，在使用 setAttribute()设置 Class 属性的时候，两种浏览器也存在同样的差异。

setAttribute("className",value);这种写法适用于 IE8.0 之前的所有 IE 版本，注意：IE8.0 也不支持"className"属性了。
setAttribute("class",value);适用于 IE8.0 以及 firefox。

【兼容办法】：

1.两种都写上：

```
//设置 header 的 class 为 classValue
var myObject = document.getElementById("header");

myObject.setAttribute("class","classValue");

myObject.setAttribute("className","classValue");
```

2.IE 和 FF 都支持 object.className，所以可以这样写：

```
var myObject = document.getElementById("header");

myObject.className="classValue";//设置 header 的 class 为 classValue
```

3.先判断浏览器类型，再根据浏览器类型采用对应的写法。

32.对象宽高赋值问题

【问题描述】：非 IE 浏览器中中类似 obj.style.height = imgObj.height 的语句无效,必须加上'px'。

【兼容办法】：给元素高度宽度附值是，统一都加上'px'，如：

```
obj.style.height = imgObj.height + 'px';
```

33.鼠标位置

【问题描述】：IE 下，even 对象有 x、y 属性，但是没有 pageX、pageY 属性；Firefox 下，even 对象有 pageX、pageY 属性，但是没有 x、y 属性；Safari、Chrome 中 x、y 属性和 pageX、pageY 都有。

【兼容办法】：使用 mX = event.x ? event.x : event.pageX;来代替。复杂点还要考虑绝对位置。

```
function getAbsPoint(e){
    var x = e.offsetLeft, y = e.offsetTop;
    while (e = e.offsetParent) {
        x += e.offsetLeft;
        y += e.offsetTop;
    }
    alert("x:" + x + "," + "y:" + y);
}
```

34.event.srcElement

【问题描述】：IE 下，event 对象有 srcElement 属性，但是没有 target 属性；其它浏览器下，even 对象有 target 属性，但是没有 srcElement 属性。

【兼容办法】：

```
var obj = event.srcElement?event.srcElement:event.target;
```

35.关于**<input type="file">**

(1) 在 safari 浏览器下的此控件没有文本框，只有一个"选取文件"的按钮，所有也没有 onblur 事件，如果在<input type="file" onblur="alert(0);">中用到了需要做特殊处理。

(2) 在 FF 浏览器下用<input type="file" name="file"> 上传文件后取 file.value 时只能去掉文件名而没有文件路径，不能实现预览的效果，可以用 document.getElementById("pic").files[0].getAsDataURL();取到加密后的路径，此路径只有在 FF 下才可以解析。

(3) 在 safari 浏览器下用<input type="file" name="file"> 上传文件后取 file.value 时只能去掉文件名而没有文件路径，不能实现预览的效果。建议使用上传后的路径预览。

36.jquery 对象是否为空

jquery 对象是否为空判断，用 length 判断一下

```
$("#hidTitle").length>0
```

原文

http://www.cnblogs.com/eagle927183/p/3478206.html?utm_source=tuicool

聊聊并发（七）——Java 中的阻塞队列

1. 什么是阻塞队列？

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

阻塞队列提供了四种处理方法:

方法\处理方式	抛出异常	返回特殊值	一直阻塞	超时退出
插入方法	add(e)	offer(e)	put(e)	offer(e, time, unit)
移除方法	remove()	poll()	take()	poll(time, unit)
检查方法	element()	peek()	不可用	不可用

- 抛出异常：是指当阻塞队列满时候，再往队列里插入元素，会抛出 IllegalStateException("Queue full")异常。当队列为空时，从队列里获取元素时会抛出 NoSuchElementException 异常 。
- 返回特殊值：插入方法会返回是否成功，成功则返回 true。移除方法，则是从队列里拿出一个元素，如果没有则返回 null
- 一直阻塞：当阻塞队列满时，如果生产者线程往队列里 put 元素，队列会一直阻塞生产者线程，直到拿到数据，或者响应中断退出。当队列空时，消费者线程试图从队列里 take 元素，队列也会阻塞消费者线程，直到队列可用。
- 超时退出：当阻塞队列满时，队列会阻塞生产者线程一段时间，如果超过一定的时间，生产者线程就会退出。

2. Java 里的阻塞队列

JDK7 提供了 7 个阻塞队列。分别是

- ArrayBlockingQueue ：一个由数组结构组成的有界阻塞队列。
- LinkedBlockingQueue ：一个由链表结构组成的有界阻塞队列。
- PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。
- DelayQueue：一个使用优先级队列实现的无界阻塞队列。
- SynchronousQueue：一个不存储元素的阻塞队列。
- LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。
- LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

ArrayBlockingQueue 是一个用数组实现的有界阻塞队列。此队列按照先进先出（FIFO）的原则对元素进行排序。默认情况下不保证访问者公平的访问队列，所谓公平访问队列是指阻塞的所有生产者线程或消费者线程，当队列可用时，可以按照阻塞的先后顺序访问队列，即先阻塞的生产者线程，可以先往队列里插入元素，先阻塞的消费者线程，可以先从队列里获取元素。通常情况下为了保证公平性会降低吞吐量。我们可以使用以下代码创建一个公平的阻塞队列：

```
ArrayBlockingQueue fairQueue = new
ArrayBlockingQueue(1000,true);
```

访问者的公平性是使用可重入锁实现的，代码如下：

```
public ArrayBlockingQueue(int capacity, boolean fair) {
       if (capacity <= 0)
           throw new IllegalArgumentException();
       this.items = new Object[capacity];
       lock = new ReentrantLock(fair);
       notEmpty = lock.newCondition();
       notFull =  lock.newCondition();
}
```

LinkedBlockingQueue 是一个用链表实现的有界阻塞队列。此队列的默认和最大长度为 Integer.MAX_VALUE。此队列按照先进先出的原则对元素进行排序。

PriorityBlockingQueue 是一个支持优先级的无界队列。默认情况下元素采取自然顺序排列，也可以通过比较器 comparator 来指定元素的排序规则。元素按照升序排列。

DelayQueue 是一个支持延时获取元素的无界阻塞队列。队列使用 PriorityQueue 来实现。队列中的元素必须实现 Delayed 接口，在创建元素时可以指定多久才能从队列中获取当前元素。只有在延迟期满时才能从队列中提取元素。我们可以将 DelayQueue 运用在以下应用场景：

- 缓存系统的设计：可以用 DelayQueue 保存缓存元素的有效期，使用一个线程循环查询 DelayQueue，一旦能从 DelayQueue 中获取元素时，表示缓存有效期到了。
- 定时任务调度。使用 DelayQueue 保存当天将会执行的任务和执行时间，一旦从 DelayQueue 中获取到任务就开始执行，从比如 TimerQueue 就是使用 DelayQueue 实现的。

队列中的 Delayed 必须实现 compareTo 来指定元素的顺序。比如让延时时间最长的放在队列的末尾。实现代码如下：

```
public int compareTo(Delayed other) {
          if (other == this) // compare zero ONLY if same object
              return 0;
           if (other instanceof ScheduledFutureTask) {
              ScheduledFutureTask x =
(ScheduledFutureTask)other;
              long diff = time - x.time;
              if (diff < 0)
                 return -1;
              else if (diff > 0)
                 return 1;
          else if (sequenceNumber < x.sequenceNumber)
                 return -1;
              else
                 return 1;
           }
           long d = (getDelay(TimeUnit.NANOSECONDS) -
                     other.getDelay(TimeUnit.NANOSECONDS));
```

```
            return (d == 0) ? 0 : ((d < 0) ? -1 : 1);
        }
```

如何实现 Delayed 接口

我们可以参考 ScheduledThreadPoolExecutor 里 ScheduledFutureTask 类。这个类实现了 Delayed 接口。首先：在对象创建的时候，使用 time 记录前对象什么时候可以使用，代码如下：

```
ScheduledFutureTask(Runnable r, V result, long ns, long
period) {
            super(r, result);
            this.time = ns;
            this.period = period;
            this.sequenceNumber =
sequencer.getAndIncrement();
}
```

然后使用 getDelay 可以查询当前元素还需要延时多久，代码如下：

```
public long getDelay(TimeUnit unit) {
            return unit.convert(time - now(),
TimeUnit.NANOSECONDS);
        }
```

通过构造函数可以看出延迟时间参数 ns 的单位是纳秒，自己设计的时候最好使用纳秒，因为 getDelay 时可以指定任意单位，一旦以纳秒作为单位，而延时的时间又精确不到纳秒就麻烦了。使用时请注意当 time 小于当前时间时，getDelay 会返回负数。

如何实现延时队列

延时队列的实现很简单，当消费者从队列里获取元素时，如果元素没有达到延时时间，就阻塞当前线程。

```
long delay = first.getDelay(TimeUnit.NANOSECONDS);
                if (delay <= 0)
                    return q.poll();
```

```
else if (leader != null)
    available.await();
```

SynchronousQueue 是一个不存储元素的阻塞队列。每一个 put 操作必须等待一个 take 操作，否则不能继续添加元素。SynchronousQueue 可以看成是一个传球手，负责把生产者线程处理的数据直接传递给消费者线程。队列本身并不存储任何元素，非常适合于传递性场景,比如在一个线程中使用的数据，传递给另外一个线程使用，SynchronousQueue 的吞吐量高于 LinkedBlockingQueue 和 ArrayBlockingQueue。

LinkedTransferQueue 是一个由链表结构组成的无界阻塞 TransferQueue 队列。相对于其他阻塞队列，LinkedTransferQueue 多了 tryTransfer 和 transfer 方法。

transfer 方法。如果当前有消费者正在等待接收元素（消费者使用 take()方法或带时间限制的 poll()方法时），transfer 方法可以把生产者传入的元素立刻 transfer（传输）给消费者。如果没有消费者在等待接收元素，transfer 方法会将元素存放在队列的 tail 节点，并等到该元素被消费者消费了才返回。transfer 方法的关键代码如下：

```
Node pred = tryAppend(s, haveData);
return awaitMatch(s, pred, e, (how == TIMED), nanos);
```

第一行代码是试图把存放当前元素的 s 节点作为 tail 节点。第二行代码是让 CPU 自旋等待消费者消费元素。因为自旋会消耗 CPU，所以自旋一定的次数后使用 Thread.yield()方法来暂停当前正在执行的线程，并执行其他线程。

tryTransfer 方法。则是用来试探下生产者传入的元素是否能直接传给消费者。如果没有消费者等待接收元素，则返回 false。和 transfer 方法的区别是 tryTransfer 方法无论消费者是否接收，方法立即返回。而 transfer 方法是必须等到消费者消费了才返回。

对于带有时间限制的 tryTransfer(E e, long timeout, TimeUnit unit)方法，则是试图把生产者传入的元素直接传给消费者，但是如果没有消费者消费该元素则等待指定的时间再返回，如果超时还没消费元素，则返回 false，如果在超时时间内消费了元素，则返回 true。

LinkedBlockingDeque 是一个由链表结构组成的双向阻塞队列。所谓双向队列指的你可以从队列的两端插入和移出元素。双端队列因为多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争。相比其他的阻塞队列，LinkedBlockingDeque 多了 addFirst，addLast，offerFirst，offerLast，peekFirst，peekLast 等方法，以 First 单词结尾的方法，表示插入，获取（peek）或移除双端队列的第一个元素。以 Last 单词结尾的方法，表示插入，获取或移除双端队列的最后一个元素。另外插入方法 add 等同于 addLast，移除方法 remove 等效

于 removeFirst。但是 take 方法却等同于 takeFirst，不知道是不是 Jdk 的 bug，使用时还是用带有 First 和 Last 后缀的方法更清楚。

在初始化 LinkedBlockingDeque 时可以设置容量防止其过渡膨胀。另外双向阻塞队列可以运用在“工作窃取”模式中。

3. 阻塞队列的实现原理

如果队列是空的，消费者会一直等待，当生产者添加元素时候，消费者是如何知道当前队列有元素的呢？如果让你来设计阻塞队列你会如何设计，让生产者和消费者能够高效率的进行通讯呢？让我们先来看看 JDK 是如何实现的。

使用通知模式实现。所谓通知模式，就是当生产者往满的队列里添加元素时会阻塞住生产者，当消费者消费了一个队列中的元素后，会通知生产者当前队列可用。通过查看 JDK 源码发现 ArrayBlockingQueue 使用了 Condition 来实现，代码如下：

```
private final Condition notFull;
private final Condition notEmpty;


public ArrayBlockingQueue(int capacity, boolean fair) {

       //省略其他代码

       notEmpty = lock.newCondition();
       notFull =  lock.newCondition();
   }


public void put(E e) throws InterruptedException {
       checkNotNull(e);
       final ReentrantLock lock = this.lock;
       lock.lockInterruptibly();
       try {
           while (count == items.length)
               notFull.await();
           insert(e);
       } finally {
           lock.unlock();
```

```
        }
}

public E take() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            while (count == 0)
                notEmpty.await();
            return extract();
  } finally {
            lock.unlock();
        }
}

private void insert(E x) {
        items[putIndex] = x;
        putIndex = inc(putIndex);
        ++count;
        notEmpty.signal();
    }
```

当我们往队列里插入一个元素时，如果队列不可用，阻塞生产者主要通过LockSupport.park(this);来实现

```
public final void await() throws InterruptedException {
            if (Thread.interrupted())
                throw new InterruptedException();
            Node node = addConditionWaiter();
            int savedState = fullyRelease(node);
            int interruptMode = 0;
```

```
            while (!isOnSyncQueue(node)) {
                LockSupport.park(this);
                if ((interruptMode =
checkInterruptWhileWaiting(node)) != 0)
                    break;
            }
            if (acquireQueued(node, savedState) &&
interruptMode != THROW_IE)
                interruptMode = REINTERRUPT;
            if (node.nextWaiter != null) // clean up if
cancelled
                unlinkCancelledWaiters();
            if (interruptMode != 0)

reportInterruptAfterWait(interruptMode);
        }
```

继续进入源码，发现调用 setBlocker 先保存下将要阻塞的线程，然后调用 unsafe.park 阻塞当前线程。

```
public static void park(Object blocker) {
        Thread t = Thread.currentThread();
        setBlocker(t, blocker);
        unsafe.park(false, 0L);
        setBlocker(t, null);
    }
```

unsafe.park 是个 native 方法，代码如下：

```
public native void park(boolean isAbsolute, long time);
```

park 这个方法会阻塞当前线程，只有以下四种情况中的一种发生时，该方法才会返回。

- 与 park 对应的 unpark 执行或已经执行时。注意：已经执行是指 unpark 先执行，然后再执行的 park。
- 线程被中断时。
- 如果参数中的 time 不是零，等待了指定的毫秒数时。
- 发生异常现象时。这些异常事先无法确定。

我们继续看一下 JVM 是如何实现 park 方法的，park 在不同的操作系统使用不同的方式实现，在 linux 下是使用的是系统方法 pthread_cond_wait 实现。实现代码在 JVM 源码路径 src/os/linux/vm/os_linux.cpp 里的 os::PlatformEvent::park 方法，代码如下：

```
void os::PlatformEvent::park() {
           int v ;
           for (;;) {
             v = _Event ;
           if (Atomic::cmpxchg (v-1, &_Event, v) == v) break ;
           }
           guarantee (v >= 0, "invariant") ;
           if (v == 0) {
           // Do this the hard way by blocking ...
           int status = pthread_mutex_lock(_mutex);
           assert_status(status == 0, status, "mutex_lock");
           guarantee (_nParked == 0, "invariant") ;
           ++ _nParked ;
           while (_Event < 0) {
           status = pthread_cond_wait(_cond, _mutex);
           // for some reason, under 2.7 lwp_cond_wait() may
return ETIME ...
           // Treat this the same as if the wait was
interrupted
           if (status == ETIME) { status = EINTR; }
           assert_status(status == 0 || status == EINTR,
status, "cond_wait");
           }
           -- _nParked ;
```

```
            // In theory we could move the ST of 0 into _Event
past the unlock(),
            // but then we'd need a MEMBAR after the ST.
            _Event = 0 ;
            status = pthread_mutex_unlock(_mutex);
            assert_status(status == 0, status,
"mutex_unlock");
            }
            guarantee (_Event >= 0, "invariant") ;
            }

    }
```

pthread_cond_wait 是一个多线程的条件变量函数，cond 是 condition 的缩写，字面意思可以理解为线程在等待一个条件发生，这个条件是一个全局变量。这个方法接收两个参数，一个共享变量_cond，一个互斥量_mutex。而 unpark 方法在 linux 下是使用 pthread_cond_signal 实现的。park 在 windows 下则是使用 WaitForSingleObject 实现的。

当队列满时，生产者往阻塞队列里插入一个元素，生产者线程会进入 WAITING (parking)状态。我们可以使用 jstack dump 阻塞的生产者线程看到这点：

```
"main" prio=5 tid=0x00007fc83c000000 nid=0x10164e000 waiting
on condition [0x000000010164d000]
   java.lang.Thread.State: WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        - parking to wait for  <0x0000000140559fe8> (a
java.util.concurrent.locks.AbstractQueuedSynchronizer$Con
ditionObject)
        at
java.util.concurrent.locks.LockSupport.park(LockSupport.j
ava:186)
        at
java.util.concurrent.locks.AbstractQueuedSynchronizer$Con
ditionObject.await(AbstractQueuedSynchronizer.java:2043)
```

```
        at
java.util.concurrent.ArrayBlockingQueue.put(ArrayBlocking
Queue.java:324)
        at
blockingqueue.ArrayBlockingQueueTest.main(ArrayBlockingQu
eueTest.java:11)
```

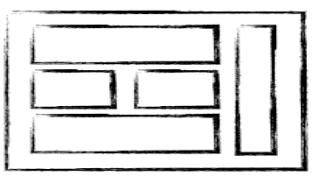
4. 参考资料

- JDK6.0 阻塞队列 API 文档
- JDK1.7 源码
- JVM Park 的 windows 实现
- JVM Park 的 linux 实现代码

原文：
http://www.infoq.com/cn/articles/java-blocking-queue?utm_source=tuicool

微服务架构解析

近日，Spring 4.0 GA 版发布，这是时隔几年后 Spring 发布的又一个重大版本，提供了诸多的新特性。Spring 4.0 是首个完全支持 Java 8 特性的框架，还提供了对云、大数据及微服务架构的支持。此外，Spring 4 还提供了对 Java EE 6 和 7、WebSocket、SockJS 以及 STOMP 及动态语言 Groovy 的支持。在新增的众多特性中，微服务架构是个很有趣的概念，它的主要作用是将功能分解到离散的各个服务当中，从而降低系统的耦合性，并提供更加灵活的服务支持。软件咨询师 James Hughes 专门撰文详细介绍了微服务的概念、作用、适用性、应用场景以及测试相关的话题。
微服务架构（MSA）是一种架构概念，旨在通过将功能分解到各个离散的服务中以实现对解决方案的解耦。你可以将其看作是在架构层次而非获取服务的类上应用很多 SOLID 原则。
从概念上来说，MSA 并不难理解，不过从实践上来说，它会引发很多问题。这些服务之间是如何通信的呢？服务之间的延迟是怎样的？如何测试服务呢？如何检测失败并对其作出响应呢？如果存在大量互相依赖的情况该如何管理部署呢？接下来的内容就将回答这些问题，我们一起来看看 MSA 是否值得学习和使用。



MONOLITHIC/LAYERED

MICRO SERVICES

解析微服务

首先我们来看看到底什么是微服务。实际上微服务本身并没有一个严格的定义，不过从很多人的反馈来看，大家都达成了这样一个共识：微服务是一种简单的应用，大概有 10 到 100 行代码。我知道使用代码行数来比较实现其实很不靠谱，因此你能理解这个意思就行，不必过分拘泥于细节。不过有一点需要注意，那就是微服务通常都是很小的，甚至是微型的。这意味着你不会在大型框架上看到很多小服务，这是不切实际的。简单与轻量级是当今的主流。诸如 Sinatra、Webbit、Finagle 与 Connect 等小型框架在将你的代码包装到一个薄薄的通信层这方面做得刚刚好。
从物理角度来说，这些服务都很小，你可以在同一台机器上运行大量服务，不必担心内存或是资源等问题。重申一遍，基于大型框架的简单库将会取得最后的胜利，你会发现对第三方库的依赖越来越少。

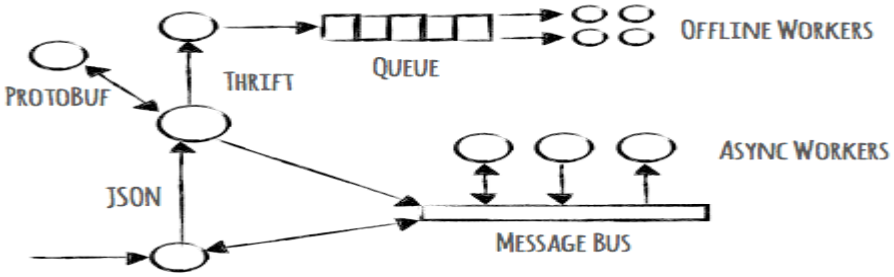
这种服务层上的解耦还提供了另外一个有趣儿的选择。我们将很多老旧应用的复杂性推到了基础设施层，不再受限于单个技术栈或是语言了。我们现在可以发挥出任何技术栈或是语言的优势。我们完全可以通过多种语言和库来构建系统，稍后我将会对此进行介绍，其实它是个双刃剑。

你不会看到任何基于微服务的架构是托管在应用服务器上的，这是关键。本质上，微服务是自我托管的，他们获取一个端口然后监听。这意味着你将失去典型的企业应用服务器所带来的很多好处，服务需要提供这些必要的功能（性能度量、监控等等）。

通信

这是个有趣的话题。服务之间该如何通信呢？这个问题是无法通过一个简单的答案解决的，甚至在单个解决方案中也是难以做到的。最基本的答案就是通过 HTTP 公开所有服务，然后以 JSON 作为数据交换格式。服务探测（一个服务是如何找到另一个服务的）可以很简单，只需将端点细节信息放到配置文件中即可（硬编码也行）。

你可能会发现在某些情况下，一个完整事务中对 JSON 负载的序列化与反序列化的代价会造成系统瓶颈。也许 JSON 并不适合，你可能需要使用别的协议，如 Protobuf 等。
不过硬编码的 URL 会导致耦合。在分层应用中，这是有意义的，不过对于基于服务的架构来说，这意味着你不能再被这种潜在的限制所约束。服务之间的某些通信可以是完全解耦的，事实上，有些服务可以随意发布事件或是数据。他们只是将其扔出去（比如说消息总线），也许有一天出现了某个服务，然后开始监听了。此外，也许系统的某些部分会以批量/离线的流程进行运作，他们可能会在几小时后才从队列中取出消息。微服务架构可以实现这种灵活性而无需修改整个架构。



COMMUNICATION BETWEEN MICRO SERVICES

监控与度量

分层解决方案的组件出现问题时不会销声匿迹，要么是编译失败，要么是遇到问题时抛出异常（除非你将抛出的异常隐藏掉了）。在基于服务的方式中，有的服务可能出现了问题，而其他服务则会很容易发现问题（特别是在 pub/sub 模型中）。这意味着我们必须要能对服务进行监控和编排。事实上，只知道服务还能用是不够的，服务是不是还能提供业务价值？还能否继续使用？它是可靠交易的瓶颈么？

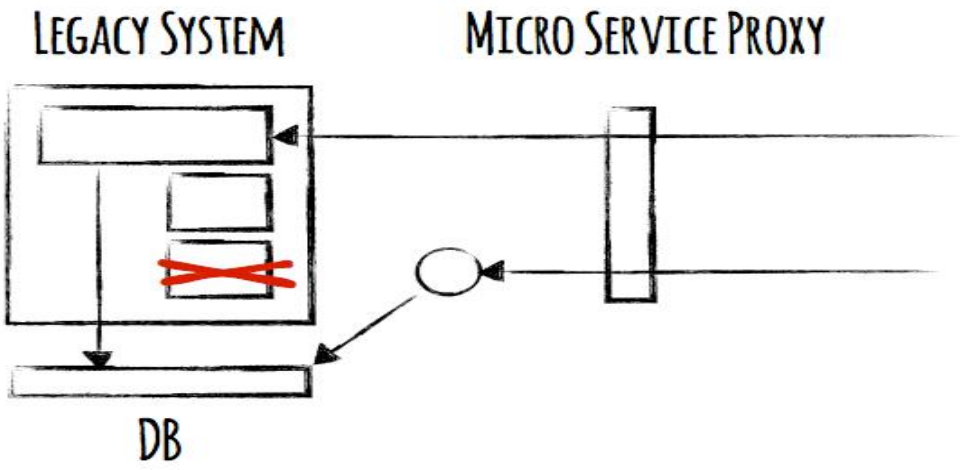
监控总是非常重要的事情，对于基于服务的架构来说更是如此，因为这时出现的失败并不容易被发现。JVM 世界中的 Metrics 与 Ostrich 等库不仅可以收集度量信息，还提供了与 Nagios 和 Ganglia 等服务的集成，可以将数据发送给他们。

测试

对于基于微服务架构的系统来说，测试服务并没有什么特殊之处，不过我这里要强调的是你不必再对每个服务使用完整的测试套件了。因为一个服务只做一件事，因此引入系统 Bug 的几率明显降低了，这要归功于基于服务的系统的天生的行为。我的意思并不是说不用做测试，而是建议你在使用过去的测试前多思考一下。

微服务架构的应用

微服务还能在大型遗留系统中大显身手。处理遗留代码是非常有风险的。对于运行了多年的系统来说，人们很有可能缺乏必要的知识来理解系统内部的运行方式。处理这种代码就像纸牌堆成的房子一样，一处出现了问题就会影响到其他地方。这些系统通常都是关键任务的系统，因此错误的代价是非常高的。你可以通过微服务方式解决这个难题。相对于直接深入到遗留代码基的做法来说，你可以编写一个小服务，它能完成你的想法（没错，你可以复制遗留代码），并且可以通过它代理服务（比如说通过 nginx/apache 或是纯代码方式）。



原文：
http://www.infoq.com/cn/news/2013/12/micro-service-architecture?utm_source=tuicool

为什么 Erlang 比 C 慢那么多倍？

Erlang 一直以慢“著称”，本文就来看看 Erlang 慢在什么地方，为什么比实现同样功能的 C 语言程序慢那么多倍。Erlang 作为一种虚拟机解释的语言，慢是当然的。不过本文从细节上分析为什么 Erlang 这种虚拟机语言会慢。

本文从 shootout benchmark[注 1]中选择了一个 Erlang 和 C 语言单核性能差距最大的例子——reverse complement[注 2]。根据 shootout 网站上给出的使用某款 64 位处理器单个核心的 benchmark 数据，Erlang 实现消耗的 CPU 时间为 19.20 秒，而 C 语言实现消耗的时间为 0.71 秒。也就是说，Erlang 实现同样的功能慢了 27 倍。本文暂不关心 Erlang 的多进程并行化的加速性能，只关心 Erlang 虚拟机单个线程执行机构的性能。

我们先来看一下这个程序要实现的功能是什么。刚好这个例子实现的功能是 shootout benchmark 目前 13 个测试中最好理解的，不需要任何数学背景和复杂的数据结构或算法，只涉及到非常简单的高中生物学知识。这个程序的功能是计算给定 DNA 序列的反向互补链。根据高中生物学，DNA 序列就是碱基对序列，碱基对是由两个互补的碱基构成的。碱基的互补关系如下所示：

1	code	meaning	complement
2	A	A	T
3	C	C	G
4	G	G	C
5	T/U	T	A
6	M	A or C	K

7	R	A or G			Y
8	W	A or T			W
9	S	C or G			S
10	Y	C or T			R
11	K	G or T			M
12	V	A or C or G		B	
13	H	A or C or T		D	
14	D	A or G or T		H	
15	B	C or G or T		V	
16	N	G or A or T or C	N		

中间那一列不用管了，也就是说左边那一列的互补碱基就是右边那一列。假设有一个序列为 ACG,那么互补序列就是 TGC,而我们要求的反向互补序列就是 CGT,要求反向的原因和 DNA 的反向转录有关。程序的输入采用 FASTA 格式，这个格式很简单，例如这个页面上的示例输入文件。FASTA 文件分为多个段落，每个段落有一个 ">" 表示的开头，这一行后面是 DNA 序列的一些信息，具体意义我们不管。接下来的行就是具体的 DNA 序列，每一行显示 60 个碱基。在具体的 FASTA

文件中碱基既可以用大写表示，也可以使用小写。要求程序输出 FASTA 格式的反向互补序列，其中“>”行照抄。比如下面这个输入文件

```
1  >ONE Homo sapiens alu
2  GGCCGGGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGGCGGA
3  TCACC
```

得到的输出文件是

```
1  >ONE Homo sapiens alu
2  GGTGATCCGCCCGCCTCGGCCTCCCAAAGTGCTGGGATTACAGGCGTGAGCCACCGCGCC
3  CGGCC
```

分析程序的需求，可以看出这个程序数据处理方面的工作在于输入文件的解析、断行拼接、计算互补序列、计算反向序列以及结果的断行输出。shootout 网站上最快的 C 语言程序在这里。这个 C 语言程序采用的方法是首先将整个文件读到一个缓冲区中，然后解析文件，找“>”行，那么这一行到下一个 “>” 行之间就是一个完整的 DNA 序列，创建新的工作线程，把这个 DNA 序列在整个缓冲区中的位置信息传递给工作线程，工作线程负责计算互补的碱基并且写入缓冲区。工作线程的主体部分很简单，维护两个指针，一头一尾，每一轮迭代都向中间挪一个位置。每一轮迭代中，计算一头一尾的互补碱基，然后交换，当头尾相

遇的时候结束。当然，如果 DNA 序列最后一行不能填满 60 个字节，工作线程还要先挪动每一行的位置，使得交换之后换行符的位置正确。这个 C 语言程序计算互补碱基采用了查表法，由于输入值就是大小写字母和换行符，所以只需要一个不大的查找表（128 字节）。

这个 C 语言程序的效率相当高，只需要一次扫描和一次写入。在最坏情况下，如果最后一行不满 60 个字符，则还需要一趟拷贝调整换行符的位置。

然后我们来看一看 shootout 网站上提供的 Erlang 程序。

这个 Erlang 程序的大致工作流程为：主进程（命名为 reader，运行 loop 函数）从 stdin 中一行一行地读取，凑齐一段完整的 DNA 序列之后，就创建一个新的进程（命名为 collector 进程）处理并打印（至 stdout）这个序列。reader 进程等待 collector 进程完成了打印之后，再继续从 stdin 中一行一行地读。collector 进程之所以叫 collector，是因为这个进程将计算反向互补序列的工作分割为几个大小均等并且带有编号的“块”，并创建若干工作进程（对应 revcomp_chunk 函数），让每个进程处理一个块，等所有工作进程都干完了之后，collector 进程对所有的块按照编号进行排序，得到正确顺序的结果，然后再将“>”行和结果都输出到 stdout。

下面简单地对 shootout 网站上提供的那个 Erlang 程序做了一些注释，我们可以看到这个程序慢在哪里：

```
% The Computer Language Benchmarks Game
% http://benchmarksgame.alioth.debian.org/
%%
%% Based on two Erlang versions contributed by
%% Vlad Balin and Fredrik Svahn.
%% contributed by Michael Pitidis
%% I/O redone by Erik Søe Sørensen

-module(revcomp).

%-compile([native, {hipe, [o3]}, inline, {inline_size, 100}]).
%-compile(export_all).

-export([main/1]).

-define(WIDTH, 60).
-define(WORKERS, 4).
-define(BUFSIZE, 4096).

main([_Args]) ->
  io:setopts([binary]),
  run_parallel(),
  halt().

%% Set up one process for reading. Transformations and printing are
%% handled asynchronously in separate processes.
run_parallel() ->
  register(reader, self()),
  reader ! go,
  loop(<< >>).

loop(Buf) ->
  case get_line() of
    eof ->
      receive go -> ok end,
      spawn(fun() -> flush(<< >>, Buf) end),
      receive go -> ok end;
    << ">", _/bytes >> = Comment ->
      receive go -> ok end,
      spawn(fun() -> flush([Comment, $\n], Buf) end),
      loop(<< >>);
    Line ->
      % Strip newline and append.
%%      S = size(Line) - 1,
%%      << Chunk:S/bytes, _ >> = Line,
      loop(<< Buf/binary, Line/binary >>)
  end.

get_line() ->
    Buf = case get(linebuf) of
            undefined -> <<>>;
            B -> B
          end,
    case binary:split(Buf, <<"\n">>) of
      [Line,Rest] ->
        put(linebuf, Rest),
        Line;
      [_] ->
        IsEOF = get(linebuf_eof) /= undefined,
        if Buf==<<>>, IsEOF ->
            eof;
           true ->
            case file:read(standard_io, ?BUFSIZE) of
              eof ->
                put(linebuf_eof, true),
                get_line();
              {ok, Data} ->
                put(linebuf, <<Buf/binary, Data/binary>>),
                get_line()
            end
        end
    end.


%% Calculate the reverse complement of Buffer, and print it.
%% Calculation is done in chunks, each assigned a separate process.
%% The results are collected, and printed in the correct order.
flush(Comment, Buffer) ->
  register(collector, self()),
  io:put_chars(reverse_complement(Buffer)),
  io:put_chars(Comment),
  unregister(collector),
  reader ! go.

%% Calculation is distributed among workers.
%% As a minor optimization, workers handle only chunks of the same size,
%% evenly divisible by ?WIDTH. The remainder is handled by the current
%% process, with a separate function.
reverse_complement(<< >>) ->
  << >>;
reverse_complement(Buffer) ->
  {Chunks, Left} = calculate_splits(size(Buffer), ?WORKERS),
  Even = start_jobs(Buffer, Chunks),
  Last = revcomp_last(Buffer, Left, << >>),
  collect(Even) ++ [Last].

start_jobs(_, 0) ->
  0;
start_jobs(Buffer, Chunks) ->
  start_jobs(Buffer, Chunks, size(Buffer), 0).

start_jobs(_, _, _, N = ?WORKERS) ->
  N;
start_jobs(Buffer, Chunk, Size, N) when Size >= Chunk ->
  new_job({fun revcomp_chunk/4, [Buffer, Size - Chunk, Size, << >>]}, N),
  start_jobs(Buffer, Chunk, Size - Chunk, N + 1).

%% Specialized function which handles even chunks.
revcomp_chunk(_, Start, Start, Acc) ->
  Acc;
revcomp_chunk(Buffer, Start, Stop, Acc) ->
  From = Stop - ?WIDTH,
  << _:From/bytes, Line:?WIDTH/bytes, _/bytes >> = Buffer,
  RC = revcomp(Line),
  revcomp_chunk(Buffer, Start, From, << Acc/binary, RC/binary >>).

%% Specialized function which handles the uneven chunk.
revcomp_last(Buffer, Stop, Acc) when Stop > ?WIDTH ->
  From = Stop - ?WIDTH,
  << _:From/bytes, Line:?WIDTH/bytes, _/bytes >> = Buffer,
  RC = revcomp(Line),
  revcomp_last(Buffer, From, << Acc/binary, RC/binary >>);
revcomp_last(Buffer, Stop, Acc) ->
  << Line:Stop/bytes, _/bytes >> = Buffer,
  RC = revcomp(Line),
  << Acc/binary, RC/binary >>.

%% Generate the reverse complement of a sequence, and append
%% a newline character.
revcomp(<< >>) ->
  << >>;
revcomp(Line) ->
  list_to_binary(lists:reverse(
      [ 10 | [ complement(C) || C <- binary_to_list(Line)]])).

calculate_splits(Size, Nodes) ->
  Tmp = Size div Nodes,
  Rem = Tmp rem ?WIDTH,
  Chunks = Tmp - Rem,
  Left = (Size rem Nodes) + (Nodes * Rem),
  {Chunks, Left}.

complement( $A ) -> $T;
complement( $C ) -> $G;
complement( $G ) -> $C;
complement( $T ) -> $A;
complement( $U ) -> $A;
complement( $M ) -> $K;
complement( $R ) -> $Y;
complement( $Y ) -> $R;
complement( $K ) -> $M;
complement( $V ) -> $B;
complement( $H ) -> $D;
complement( $D ) -> $H;
complement( $B ) -> $V;
complement( $a ) -> $T;
complement( $c ) -> $G;
complement( $g ) -> $C;
complement( $t ) -> $A;
complement( $u ) -> $A;
complement( $m ) -> $K;
complement( $r ) -> $Y;
complement( $y ) -> $R;
complement( $k ) -> $M;
complement( $v ) -> $B;
complement( $h ) -> $D;
complement( $d ) -> $H;
complement( $b ) -> $V;
complement( $N ) -> $N;
complement( $S ) -> $S;
complement( $W ) -> $W;
complement( $n ) -> $N;
complement( $s ) -> $S;
complement( $w ) -> $W.

%% Parallel helpers.
new_job({Fun, Args}, N) ->
  spawn(fun() -> collector ! {N, apply(Fun, Args)} end).

collect(N) -> collect(N, []).
collect(0, Results) -> [ R || {_, R} <- lists:keysort(1, Results) ];
collect(N, Results) -> receive {K, R} -> collect(N-1, [{K, R} | Results]) end.
```

凑齐了一段完整序列之后，创建新的 collector 进程处理

新读入的一行仍然属于 DNA 序列，因此把这一行和之前已经读到的拼接在一起，去掉换行符，形成新的 binary，然后再进入下一轮读取行的循环。
注意这里的拼接操作，会分配新的 sub binary，还有可能会有预分配的操作。
Line 的数据会被复制一次。
因此在这个位置，综合起来整个文件都会被拷贝一遍，并且会执行等同于输入文件行数次数的 sub binary 分配操作。

这里实现的 get_line 的作用是每次从输入流中获得一行数据。采用的是类似收取网络数据包的方法：一次读入固定大小的缓冲区，然后找换行符，然后剩下的不足以构成一行的和下一次收入的数据拼接。
这里的 split 函数开销很大，因为会调用开销较大的 binary:matches 函数（binary:match 函数的开销要小一些）。在 profiling 的时候会发现虚拟机中的 binary_matches_3 函数耗时比较多。
在这个位置，综合起来相当于会对整个文件进行一次扫描查找换行符。

这里的追加也是，综合起来相当于把整个文件的数据又进行了一次拷贝。

这个创建工作进程的方式很聪明，按照从后往前的顺序分块，因此分块的顺序和进程创建（编号）的顺序实际上是逆序，因此只要每一个分块里面都逆了那么整个拼起来的顺序也是逆的。

这里也是，在分块内部，从后向前的顺序开始一行一行地取数据，调用 revcomp 对单独一行进行反向互补操作。最后拼接起来的结果也是整个分块被反向互补的结果。
这个递归调用同样也要进行 binary 追加的操作。因此这里也涉及到 sub binary 的分配，并且每次追加都要复制一行的数据（RC）。
因此在这个位置，综合起来整个文件都会被拷贝一遍，并且会执行等同于输入文件行数次数的 sub binary 分配操作。

这一块是整个程序真正的热点所在。revcomp 调用一次对 Line 进行操作，也就是固定长度的 binary（在这个例子中为 60 字节）。revcomp 需要对 Line 中的每一个字节进行操作，因此首先将 Line 转换为一个列表，然后通过列表 comprehension 的操作，取出列表中的每一个元素，调用 complement 函数进行转换，然后生成新的列表，再在列表头部加上换行符。然后针对新构造的列表进行 reverse 操作，最后再将列表转换回 binary。
这个操作显然很低效。首先 binary_to_list 需要，一次性分配列表所需的空间，并且将 binary 中的内容拷贝到列表中。这种拷贝不能发挥总线带宽的任何优势，因为只能一个字节一个字节地拷贝，每次还需要维护列表中的数据结构。然后是构造求补之后的列表，这个列表需要针对 binary 中的每一个字节进行一次分配。接下来是列表反向，lists:reverse 需要扫描两次列表，一次计算长度，一次修改指针。我们要特别注意列表的扫描，因为列表中每一项需要耗费 2 个 Eterm，64 位机上就是 16 字节，因此通常 4 个列表项就要废掉一条 cacheline。对于这里这种需要处理大量单个字节的任务来说，列表太重了。最后的 list_to_binary，又是两轮扫描，一轮算大小，一轮拷贝。
综合起来看，整个输入文件在这里要被翻来覆去地拷贝好几轮了，而且 cache 利用效率低。

这一块虽然也有列表操作，但是不用担心，因为这里的列表规模很小。这里要处理的列表元素数目只有工作进程的数目，在这个例子里面为 4。

然后我就按照之前说的那个 C 语言程序的思路，另外写了一个 Erlang 程序，对 binary 的操作做了一些优化，主要使用了 binary comprehension 来缓解上述程序中的热点，不过运行时间提升并不大，大概快了 10% 吧。为啥还是慢呢？下面来细细分析。先看代码吧：

```
-module(revcomp_opt).

-export([main/1]).

-define(WIDTH, 60).
-define(BUFSIZE, 1024*1024).
-define(NEWLINECHAR, 10).

-record(scan_state,
        {current_state = search_header_begin, %
search_header_end,search_is_over
         current_header = <<>>,
         current_body = [<<>>]}).

main([_Args]) ->
    io:setopts([binary]),
    InitState = #scan_state{
        current_state = search_header_begin,
        current_header = <<>>,
        current_body = []
    },
    % fprof:trace(start, "revcomp_opt_small.trace"),
    read_and_process_file(<<>>, InitState, []),
    % fprof:trace(stop),
    halt().

read_and_process_file(Buf, State, JobList) ->
    case State#scan_state.current_state of
        search_header_begin ->
            % 寻找">"
            case binary:match(Buf, <<">">>) of
                nomatch ->
                    % 继续搜索，新添加的内容应该放在 body 中
                    NState = State#scan_state{
                        current_body=[Buf |
State#scan_state.current_body]
                    },
                    get_new_chunk(NState, JobList);
                {HeaderStartPos, _Length} ->
```

```
38                      % 找到了">", 说明要开启新的行，并且结束之前的
body
39                      % 创建新的进程处理 header/body
40                      {PreviousBody, BufLeft} = split_binary(Buf,
HeaderStartPos),
41                      NState = State#scan_state{
42                          current_state = search_header_end,
43                          current_header = <<>>,
44                          current_body = []
45                      },
46                      case State#scan_state.current_header of
47                          <<>> ->
48                              % 表示是第一次进来,继续找 header 的结
尾 “\n”
49                              read_and_process_file(BufLeft, NState,
JobList);
50                          _ ->
51                              % 形成了新的完整 body，创建新的进程处
理
52                              NewJob = start_revcomp_job(
53
State#scan_state.current_header,
54                                                  [PreviousBody |
State#scan_state.current_body],
55                                                  self()),
56                              read_and_process_file(BufLeft,
57                                                    NState,
58                                                    [NewJob |
JobList])
59                      end
60              end;
61          search_header_end ->
62              % 寻找 ">" 行结尾的 "\n"
63              case binary:match(Buf, <<"\n">>) of
64                  nomatch ->
65                      % 继续搜索，如果没找到，则把整个 Buf 追加到
current_header 中
66                      NState = State#scan_state{
67                          current_header =
68
<<(State#scan_state.current_header)/binary,
69                                    Buf/binary>>
70                      },
71                      get_new_chunk(NState, JobList);
```

```
 72                  {HeaderEndPos, _Length} ->
 73                      % 找到了 header 行的"\n"，说明 header 已经全了，
要开始构建 body
 74                      {PreviousHeader, BufLeft} = split_binary(Buf,
HeaderEndPos),
 75                      NState = State#scan_state{
 76                          current_state = search_header_begin,
 77                          current_header =
 78
<<(State#scan_state.current_header)/binary,
 79                                  PreviousHeader/binary>>
 80                      },
 81                      read_and_process_file(BufLeft, NState,
JobList)
 82              end;
 83          search_is_over ->
 84              % 文件已经扫描完了
 85              case State#scan_state.current_header of
 86                  <<>> ->
 87                      AllJobs = JobList;
 88                  _ ->
 89                      NewJob =
start_revcomp_job(State#scan_state.current_header,
 90
State#scan_state.current_body,
 91                                                  self()),
 92                      AllJobs = [NewJob | JobList]
 93              end,
 94              % 收集进程的处理结果
 95              collect_revcomp_jobs(lists:reverse(AllJobs))
 96      end.
 97
 98 get_new_chunk(State, JobList) ->
 99      case file:read(standard_io, ?BUFSIZE) of
100          eof ->
101              NState = State#scan_state{current_state =
search_is_over},
102              read_and_process_file(<<>>, NState, JobList);
103          {ok, Chunk} ->
104              read_and_process_file(Chunk, State, JobList)
105      end.

106
107 collect_revcomp_jobs([]) ->
```

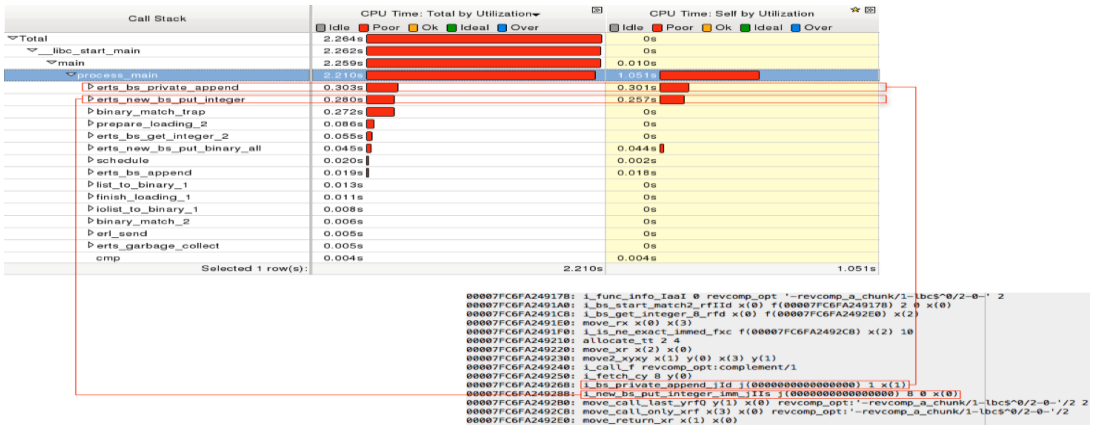
```
108     ok;
109 collect_revcomp_jobs([Job | Rest]) ->
110     receive
111         {Job, HeaderBuf, RevCompBodyPrint} ->
112             erlang:display(Job),
113             file:write(standard_io, [HeaderBuf, ?NEWLINECHAR,
RevCompBodyPrint])
114     end,
115     collect_revcomp_jobs(Rest).
116
117 start_revcomp_job(HeaderBuf, BodyBufList, Master) ->
118     spawn(fun() -> revcomp_job(HeaderBuf, BodyBufList, Master)
end).
119
120 revcomp_job(HeaderBuf, BodyBufList, Master) ->
121     RevCompBody = << <<(revcomp_a_chunk(ABuf))/binary>> ||
122                      ABuf <- BodyBufList >>,
123     RevCompBodyPrint = revcomp_chunk_printable(<<>>, RevCompBody),
124     Master ! {self(), HeaderBuf, RevCompBodyPrint}.
125
126 revcomp_a_chunk(Chunk) ->
127     Complement = << <<(complement(Byte))>> ||
128                     <<Byte>> <= Chunk, Byte =/= ?NEWLINECHAR >>,
129     % 翻转
130     ComplementBitSize = bit_size(Complement),
131     <<X:ComplementBitSize/integer-little>> = Complement,
132     ReversedComplement = <<X:ComplementBitSize/integer-big>>,
133     ReversedComplement.
134     %list_to_binary(lists:reverse([complement(C) || C <-
binary_to_list(Chunk), C=/= ?NEWLINECHAR])).
135
136 revcomp_chunk_printable(Acc, Rest) when byte_size(Rest) >= ?WIDTH ->
137     <<Line:?WIDTH/binary, Rest0/binary>> = Rest,
138     revcomp_chunk_printable(<<Acc/binary,
Line/binary, ?NEWLINECHAR>>, Rest0);
139 revcomp_chunk_printable(Acc, Rest) ->
140     <<Acc/binary, Rest/binary, ?NEWLINECHAR>>.
141
142 complement( $A ) -> $T;
143 % 同前一个 Erlang 程序，以下省略 complement 的其他子句。
```

这个程序每次从输入文件中读取一块数据，这里设置为 1M 字节。读文件的循环实际上是一个简单的状态机，通过搜索“>”字符和换行符变换状态。初始状态搜索“>”，搜索成功要么说明刚开始处理文件，要么说明已经凑齐了完整的 DNA 序列，于是进入找换行符的状态。因此这个搜索的过程综合起来看只会对输入文

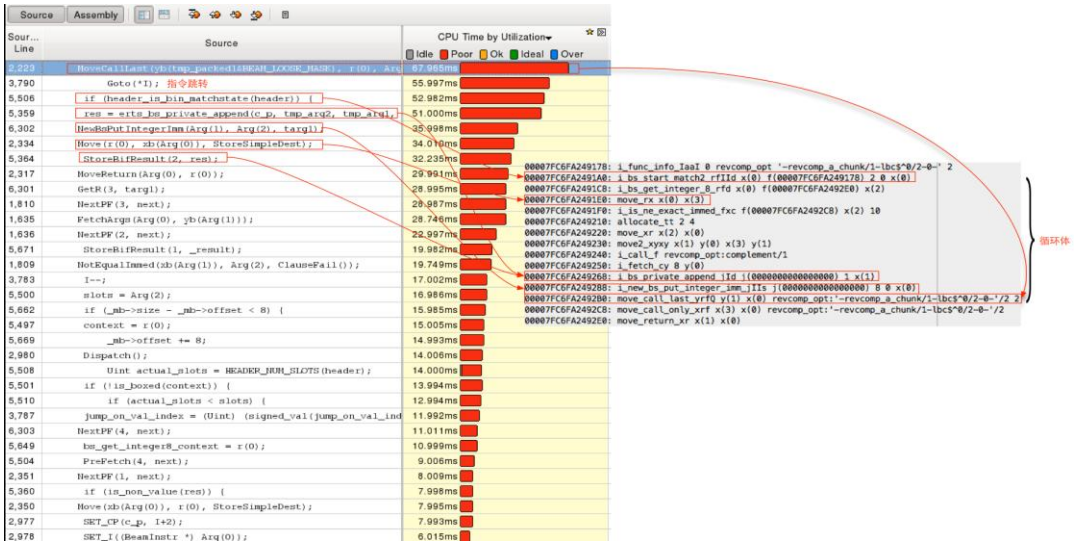
件进行一次扫描，而且使用了效率较高的 binary:match/2 函数，几乎相当于线性搜索。此外，由于搜索是针对缓冲块进行的，而不是针对每一行进行的，因此调用这个函数的频率也很低，降低了 binary:match 系列函数的启动开销（binar:match 函数每一次调用的时候都会对 pattern 做一次编译解析，这个启动开销在调用频繁的时候不可忽略）。读文件进程在扫描过程中的状态数据存放在 scan_state 记录中。其中 current_body 部分是一个列表。使用列表的目的是为了避免拼接 binary 产生的额外拷贝开销。

凑齐一段完整的 DNA 序列之后，创建一个工作进程（revcomp_job）处理这个完整的序列。以上读文件的循环传递给工作进程的 DNA 序列中是带换行符的，因此这个程序把换行符的处理从读文件的进程挪到了工作进程。工作进程需要对 current_body 列表中每一项表示的 binary 做反向互补操作。这个反向互补操作就是通过 revcomp_a_chunk 函数进行的。这个函数通过 binary comprehension 操作，从 binary 中逐个取出字节，然后调用 complement 函数计算互补碱基，并填入结果 binary。这个 comprehension 操作会预估好最终 binary 的大小，一次性做好分配，然后直接写入。针对原始数据只会进行一次扫描。计算完互补碱基之后，接下来是一个快速的 binary 翻转操作。这个翻转是从网上找到的[注 3]，非常巧妙高效，首先将要翻转的 binary 以一个巨大的小尾顺序大整数读入，然后再将其以大尾顺序的方式写入一个新的 binary，那么得到的这个 binary 就是原 binary 翻转后的结果。整个翻转操作是在 ERTS 内部通过 C 语言实现的，效率非常高。

很明显，revcomp_a_chunk 函数就是整个程序的热点所在，通过 profiling 也可以看出这一点。下图展示的是针对一个较小的输入文件的 profiling 结果。在虚拟机中，主要耗时的显然是 process_main 函数了，因为这是虚拟机的引擎函数，整个虚拟机的逻辑都在这里，说明在执行这段程序的时候，有 2.21 秒的 CPU 时间都在运行 Erlang 虚拟机指令。图的右下角的截图是 revcomp_a_chunk 函数中 binary comprehension 部分的虚拟机指令。这些指令是虚拟机加载之后实际执行的指令。可以看出，process_main 的 stack trace 中耗时最多的两个函数分别对应了其中的两条指令。也就是说，在 process_main 函数耗费的 2.210 秒中，有 0.303 秒中耗费在实现 i_bs_private_append_jld 指令的 erts_bs_private_append 函数，还有 0.280 秒的 CPU 时间用在实现 i_new_bs_put_integer_imm_jIIs 指令的 erts_new_bs_put_integer 函数。根据这篇博文的介绍，这两条指令中的前一条的作用是维护可写 binary 数据结构，后一条的作用是填入数据。



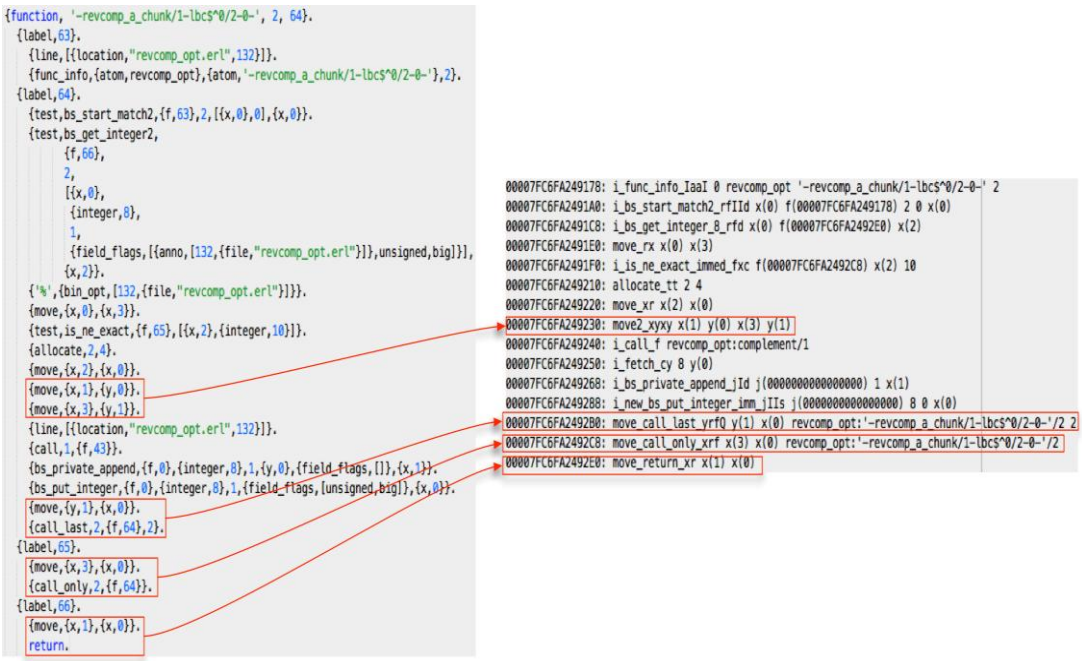
从图中还可以看出，有 1.051 秒的 CPU 时间耗在 process_main() 函数本身的语句，所以我们深入 process_main() 看一下里面耗时的语句有哪些。下图根据语句的耗时对语句进行了排序。明显可以看出，那些耗时很多的语句都和 binary comprehension 生成的这个大循环中的指令有关系。



从上面的两张图可以看出，这些开销都是省不掉的了。因为 Erlang 作为一种通用的语言，其提供的数据结构具有一定的通用性，所以很难做到针对某一个任务特别优化。比如说我们这个例子中循环操纵一个缓冲区中每一个字节并写入另外一个缓冲区的操作，就需要通过上面这些虚拟机指令来实现：读取一个字节，需要通过 i_bs_get_integer_8_rfd 指令，写入一个字节需要 i_bs_private_append_jld 和 i_new_bs_put_integer_imm_jIIs 两条指令来实现，而这些指令实现的是通用的取值和写值的操作，因此指令本身的实现涉及到很多函数调用、参数检查和数据结构维护的操作，这必然比纯 C 语言的字节拷贝要慢多了。而且虚拟机指令跳转本身也是有很大开销的，从上图中的 Goto 语句和一些 NextPF 语句就可以看出来。

那么碰到这种任务应该怎么办呢？解决方法就是，我们不要被语言所束缚，该用 C 语言的地方还是用 C 语言吧，发挥各种语言自身的优势，而不要用一种语言的弱点去和另一种语言的强项作比较，有得必有失。Erlang 也是很体贴地提供了诸如 NIF 机制让我们可以用 C 语言实现一些只有 C 语言才能高效完成的任务。比如说在这个程序中，用 C 语言来实现 revcomp_a_chunk 函数就是很好的选择。

本文到这里应该结束了，不过下面我还要啰嗦一些关于 Erlang 有意思的地方。我们刚才提到，虚拟机指令分发本身也是高开销的操作，因为指令分发意味着跳转，而 CPU 最擅长的是顺序执行，跳转会破坏 CPU 分支预测的优化，因此很多语言虚拟机的一大优化就是尽量减少指令的分发，也就是尽可能地在一条指令中执行更多的任务。比如说下面的图还是以 binary comprehension 的那一段为例，左侧是 beam 汇编码，这是编译器从原始码直接生成的，右侧是虚拟机加载优化之后实际运行的指令：



从图中可以看出，左侧有一些常见的指令组合在右侧被优化为一条指令了，可以在一定程度上减少指令分发。

另外，关于 complement 函数，这个函数有 32 个子句，实际上实现了一个映射关系，那么 Erlang 会怎么处理这个函数呢？显然也是有大大的优化滴。放心，我们聪明的 Erlang 不会笨笨地每一次调用的时候都顺序查找这 32 条子句。先看一下编译器生成的 complement 汇编码：

```
{function, complement, 1, 43}.
  {label,42}.
    {line,[{location,"revcomp_opt.erl",146}]}.
    {func_info,{atom,revcomp_opt},{atom,complement},1}.
  {label,43}.
    {test,is_integer,{f,42},[{x,0}]}.
    {select_val,{x,0},
                {f,42},
                {list,[{integer,114},
                       {f,44},
                       {integer,82},
                       {f,44},
                       % 省略...
                       {integer,84},
                       {f,58}]}}.
  {label,44}.
    {move,{integer,89},{x,0}}.
    return.
  {label,45}.
    {move,{integer,87},{x,0}}.
    return.
  % 省略...
  {label,58}.
    {move,{integer,65},{x,0}}.
    return.
```

可以看出，调用这个函数的时候，主要起作用的是 select_val 指令，这条指令根据输入值，选择一个跳转标签，比如输入 114（即“r”的 ASCII 码），跳转到标签 44 的位置，可以看到在标签 44 处返回了整数 89，即“Y”对应的 ASCII

如果我们查看 process_main() 函数对 select_val 的实现，会发现这条指令的实现采用了二分搜索，因此比线性搜索要快。但是 Erlang 就满足于此了吗？再看看虚拟机加载之后生成的实际指令：

```
00007FC6FA248D98: i_func_info_IaaI 0 revcomp_opt complement 1
00007FC6FA248DC0: i_jump_on_val_rfII x(0) f(00007FC6FA248D98) 57 65 f(
00007FC6FA248FD8) f(00007FC6FA248FC8) f(00007FC6FA249048) f(00007FC6FA249038) f
(00007FC6FA248D98) f(00007FC6FA248D98) f(00007FC6FA249068) f(00007FC6FA249058)
f(00007FC6FA248D98) f(00007FC6FA248D98) f(00007FC6FA249018) f(00007FC6FA248D98)
 f(00007FC6FA249028) f(00007FC6FA249008) f(00007FC6FA248D98) f(00007FC6FA248D98
) f(00007FC6FA248D98) f(00007FC6FA248FA8) f(00007FC6FA248FE8) f(
00007FC6FA249088) f(00007FC6FA249088) f(00007FC6FA249078) f(00007FC6FA248FB8) f
(00007FC6FA248D98) f(00007FC6FA248FF8) f(00007FC6FA248D98) f(00007FC6FA248D98)
f(00007FC6FA248D98) f(00007FC6FA248D98) f(00007FC6FA248D98) f(00007FC6FA248D98)
 f(00007FC6FA248D98) f(00007FC6FA248FD8) f(00007FC6FA248FC8) f(00007FC6FA249048
) f(00007FC6FA249038) f(00007FC6FA248D98) f(00007FC6FA248D98) f(
00007FC6FA249068) f(00007FC6FA249058) f(00007FC6FA248D98) f(00007FC6FA248D98) f
(00007FC6FA249018) f(00007FC6FA248D98) f(00007FC6FA249028) f(00007FC6FA249008)
f(00007FC6FA248D98) f(00007FC6FA248D98) f(00007FC6FA248D98) f(00007FC6FA248FA8)
 f(00007FC6FA248FE8) f(00007FC6FA249088) f(00007FC6FA249088) f(00007FC6FA249078
) f(00007FC6FA248FB8) f(00007FC6FA248D98) f(00007FC6FA248FF8)
00007FC6FA248FA8: move_return_cr 89 x(0)
00007FC6FA248FB8: move_return_cr 87 x(0)
00007FC6FA248FC8: move_return_cr 86 x(0)
00007FC6FA248FD8: move_return_cr 84 x(0)
00007FC6FA248FE8: move_return_cr 83 x(0)
00007FC6FA248FF8: move_return_cr 82 x(0)
00007FC6FA249008: move_return_cr 78 x(0)
00007FC6FA249018: move_return_cr 77 x(0)
00007FC6FA249028: move_return_cr 75 x(0)
00007FC6FA249038: move_return_cr 72 x(0)
00007FC6FA249048: move_return_cr 71 x(0)
00007FC6FA249058: move_return_cr 68 x(0)
00007FC6FA249068: move_return_cr 67 x(0)
00007FC6FA249078: move_return_cr 66 x(0)
00007FC6FA249088: move_return_cr 65 x(0)
```

怎么样？厉害吧！加载器把 select_val 给替换掉了，生成了一条更快速的指令 i_jump_on_val_rfII，这条指令的参数是一个跳转表，通过输入参数可以直接得到跳转表中的索引，从而得到跳转地址。这一下我们再也不用担心 complement 的效率了，这种函数会被优化为常量时间，实际上已经可以匹配本文开头提到的 C 语言程序采用的查找表的算法了。

当然，熟悉编译的同学可能会觉得这些都是编译领域中常用的优化技术，不过作为编译领域的外行，我还是觉得 Erlang 在编译器和虚拟机的配合优化上下了很大的功夫。

[注 1] The Computer Language Benchmarks Game
http://benchmarksgame.alioth.debian.org/

[注 2]
http://benchmarksgame.alioth.debian.org/u64/benchmark.php?test=revcomp&lang=all&data=u64

[注 3]
http://sifumoraga.blogspot.com/2010/12/reversing-binary-object-in-erlang.html

原文：
http://www.cnblogs.com/zhengsyao/p/why_is_erlang_slow_revcomp_example.html?utm_source=tuicool

PHP 真正多线程的使用

PHP 5.3 以上版本，使用 pthreads PHP 扩展，可以使 PHP 真正地支持多线程。多线程在处理重复性的循环任务，能够大大缩短程序执行时间。

我之前的文章中说过，大多数网站的性能瓶颈不在 PHP 服务器上，因为它可以简单地通过横向增加服务器或 CPU 核数来轻松应对（对于各种云主机，增加 VPS 或 CPU 核数就更方便了，直接以备份镜像增加 VPS，连操作系统、环境都不用安装配置），而是在于 MySQL 数据库。如果用 MySQL 数据库，一条联合查询的 SQL，也许就可以处理完业务逻辑，但是，遇到大量并发请求，就歇菜了。如果用 NoSQL 数据库，也许需要十次查询，才能处理完同样地业务逻辑，但每次查询都比 MySQL 要快，十次循环 NoSQL 查询也许比一次 MySQL 联合查询更快，应对几万次/秒的查询完全没问题。如果加上 PHP 多线程，通过十个线程同时查询 NoSQL，返回结果汇总输出，速度就要更快了。我们实际的 APP 产品中，调用一个通过用户喜好实时推荐商品的 PHP 接口，PHP 需要对 BigSea NoSQL 数据库发起 500~1000 次查询，来实时算出用户的个性喜好商品数据，PHP 多线程的作用非常明显。

PHP 扩展下载：https://github.com/krakjoe/pthreads
PHP 手册文档：http://php.net/manual/zh/book.pthreads.php

1、扩展的编译安装(Linux），编辑参数 --enable-maintainer-zts 是必选项：

```
cd /Data/tgz/php-5.5.1
./configure --prefix=/Data/apps/php
--with-config-file-path=/Data/apps/php/etc --with-mysql=/Data/apps/mysql
--with-mysqli=/Data/apps/mysql/bin/mysql_config --with-iconv-dir
--with-freetype-dir=/Data/apps/libs --with-jpeg-dir=/Data/apps/libs
--with-png-dir=/Data/apps/libs --with-zlib --with-libxml-dir=/usr --enable-xml
--disable-rpath --enable-bcmath --enable-shmop --enable-sysvsem
--enable-inline-optimization --with-curl --enable-mbregex --enable-fpm
--enable-mbstring --with-mcrypt=/Data/apps/libs --with-gd
--enable-gd-native-ttf --with-openssl --with-mhash --enable-pcntl
--enable-sockets --with-xmlrpc --enable-zip --enable-soap --enable-opcache
--with-pdo-mysql --enable-maintainer-zts
make clean
make
make install

unzip pthreads-master.zip
cd pthreads-master
/Data/apps/php/bin/phpize
./configure --with-php-config=/Data/apps/php/bin/php-config
```

make
make install

vi /Data/apps/php/etc/php.ini

添加：
```
extension = "pthreads.so"
```

2、给出一段 PHP 多线程、与 For 循环，抓取百度搜索页面的 PHP 代码示例：

view plainprint?

```
1.  <?php
2.    class test_thread_run extends Thread
3.    {
4.        public $url;
5.        public $data;
6.
7.        public function __construct($url)
8.        {
9.            $this->url = $url;
10.       }
11.
12.       public function run()
13.       {
14.           if(($url = $this->url))
15.           {
16.               $this->data = model_http_curl_get($url);
```

```
17.             }
18.         }
19.     }
20.
21.     function model_thread_result_get($urls_array)
22.     {
23.         foreach ($urls_array as $key => $value)
24.         {
25.             $thread_array[$key] = new test_thread_run($value["
    url"]);
26.             $thread_array[$key]->start();
27.         }
28.
29.         foreach ($thread_array as $thread_array_key => $thread_
    array_value)
30.         {
31.             while($thread_array[$thread_array_key]->isRunning()
    )
32.             {
33.                 usleep(10);
34.             }
35.             if($thread_array[$thread_array_key]->join())
```

```
36.             {
37.                 $variable_data[$thread_array_key] = $thread_ar
    ray[$thread_array_key]->data;
38.             }
39.         }
40.         return $variable_data;
41.     }
42.
43.     function model_http_curl_get($url,$userAgent="")
44.     {
45.         $userAgent = $userAgent ? $userAgent : 'Mozilla/4.0 (c
    ompatible; MSIE 7.0; Windows NT 5.2)';
46.         $curl = curl_init();
47.         curl_setopt($curl, CURLOPT_URL, $url);
48.         curl_setopt($curl, CURLOPT_RETURNTRANSFER, 1);
49.         curl_setopt($curl, CURLOPT_TIMEOUT, 5);
50.         curl_setopt($curl, CURLOPT_USERAGENT, $userAgent);
51.         $result = curl_exec($curl);
52.         curl_close($curl);
53.         return $result;
54.     }
55.
```

```
56.    for ($i=0; $i < 100; $i++)
57.    {
58.        $urls_array[] = array("name" => "baidu", "url" => "htt
   p://www.baidu.com/s?wd=".mt_rand(10000,20000));
59.    }
60.
61.    $t = microtime(true);
62.    $result = model_thread_result_get($urls_array);
63.    $e = microtime(true);
64.    echo "多线程：".($e-$t)."\n";
65.
66.    $t = microtime(true);
67.    foreach ($urls_array as $key => $value)
68.    {
69.        $result_new[$key] = model_http_curl_get($value["url"]);
70.    }
71.    $e = microtime(true);
72.    echo "For 循环：".($e-$t)."\n";
73. ?>
```

原文：http://blog.s135.com/pthreads/?utm_source=tuicool

R 语言教程：写给高级入门者的数据打理攻略(1)

学习如何添加 column、计算总和、对结果排序以及数据改造。

强大的能力在带来责任之外，也给我们增添了恼人的复杂性——这一点在 R 语言身上就表现得淋漓尽致。作为专门针对统计计算工作的开源项目，R 语言出色的调查、处理以及分析实力足以把数据驾驭得服服贴贴。然而由于其语法有时候太过复杂，入门者们可能发现自己在掌握了基础知识之后很难进一步提升个人技能水平。

如果大家还未完全熟悉 R 语言、甚至不能轻松利用它实现最基本的处理任务，我建议各位先查阅其它指导文章、帮助自己积累对 R 语言的认识。但如果大家已经拥有一定的背景知识，希望能够进一步提升自己的开发技能——或者单纯只是想看看 R 语言如何完成文章中罗列的四项任务——那么请跟着我继续阅读。

我创建了一套样本数据集，其中包含最近三年以来苹果、谷歌以及微软公司的营收与利润数额。（统计数据来自这三家公司公布的财报结果，其中 fy 代表财年。）如果大家想一步步跟随本文进行尝试，那么请将下列内容输入（或者直接复制加粘贴）到自己的 R 终端窗口当中：

```
1. fy <- c(2010, 2011, 2012, 2010, 2011, 2012, 2010, 2011, 2012)
2. company <- c("Apple", "Apple", "Apple", "Google", "Google", "Googl
   e", "Microsoft",
3. "Microsoft", "Microsoft")
4. revenue <- c(65225, 108249, 156508, 29321, 37905, 50175, 62484, 6994
   3, 73723)
5. profit <- c(14013, 25922, 41733, 8505, 9737, 10737, 18760, 23150, 169
   78)
```

```
companiesData <- data.frame(fy, company, revenue, profit)
```

以上代码将创建出如下所示的数据框，所有变量都保存在“companiedsData”当中：

	fy	company	revenue	profit
1	2010	Apple	65225	14013
2	2011	Apple	108249	25922
3	2012	Apple	156508	41733
4	2010	Google	29321	8505
5	2011	Google	37905	9737
6	2012	Google	50175	10737
7	2010	Microsoft	62484	18760
8	2011	Microsoft	69943	23150
9	2012	Microsoft	73723	16978

（如果大家没有为各行命名，那么 R 会为其自动添加行数。）

如果大家想在数据框中运行 str()函数来查看其结构，则会看到其中的“year”被当作单独的数字来处理，而无法代表应有的“年”这一含义：

```
str(companiesData)
'data.frame': 9 obs. of 4 variables:
$ fy : num 2010 2011 2012 2010 2011 ...
$ company: Factor w/ 3 levels "Apple","Google",..: 1 1 1 2
2 2 3 3 3
$ revenue: num 65225 108249 156508 29321 37905 ...
$ profit : num 14013 25922 41733 8505 9737 ...
```

我可能希望把自己的数据按年度进行分组，但大家别误会——我并不打算针对时间进行特殊分析。因此，我会将 fy 数列转化为一个包含有 Rcategory（称之为 factor）的 column 以取代日期，如以下命令所示：

```
companiesData$fy <- as.factor(companiesData$fy)
```

现在我们已经做好了各项准备工作。

向现有数据框中添加 column

在 R 语言中，最简单的任务执行方式是向基于一个或多个 column 的数据框中添加新的 column。大家可能希望添加几项现有 column 以获取平均值或者根据各行现有数据计算出某项特定“result”。

在 R 语言中我们可以通过多种方式实现这一目标。对于这样一项简单的任务，某些做法显得有些太过复杂——但请大家记住我的建议，对于那些需要处理更高难任务的高级用户来说，看似复杂的做法却往往能收到奇效。

语法一：为新 column 简单创建一个变量名称，并将其添加到计算公式中作为赋值——举例来说，我们希望在新的 column 中计算两个现有 column 的总和：

```
dataFrame$newColumn <- dataFrame$oldColumn1 + dataFrame$oldColumn2
```

大家可能已经猜到了，这个新增 colume 名为“newColumn”，其数值为 oldColumn1 与 oldColumn2 各行数值的总和。

我们的这套示例数据框名为“data”，大家可以通过将利润除以营收再乘以 100 的方式添加一个“margin”（利润率）column：

```
companiesData$margin <- (companiesData$profit / companiesData$revenue) * 100
```

运行结果如下：

	fy	company	revenue	profit	margin
1	2010	Apple	65225	14013	21.48409
2	2011	Apple	108248	25922	23.94664
3	2012	Apple	156508	41733	26.66509

	fy	company	revenue	profit	margin
4	2010	Google	29321	8505	29.00651
5	2011	Google	37905	9737	25.68790
6	2012	Google	50175	10737	21.39910
7	2010	Microsoft	62484	18760	30.02369
8	2011	Microsoft	69943	23150	33.09838
9	2012	Microsoft	73723	16978	23.02945

哇哦——大家可以看到，margin 列中数字的小数点后取值有点太夸张了。

我们可以利用 round()函数让计算结果只保留小数点后一位；round()的格式为：

round(number(s)这里填写大家想要保留的小数点位数，数字会自动进行四舍五入)

此，我们打算为 margin 列中的数字保留小数点后一位：

```
companiesData$margin <- round(companiesData$margin, 1)
```

下面就是我们得到的最新结果：

	fy	company	revenue	profit	margin
1	2010	Apple	65225	14013	21.5
2	2011	Apple	108248	25922	23.9
3	2012	Apple	156508	41733	26.7
4	2010	Google	29321	8505	29.0
5	2011	Google	37905	9737	25.7
6	2012	Google	50175	10737	21.4
7	2010	Microsoft	62484	18760	30.0
8	2011	Microsoft	69943	23150	33.1
9	2012	Microsoft	73723	16978	23.0

语法二：R 语言的 transform()函数是我们达成目标的另一条途径。以下为 transform()的基本语法：

```
dataFrame <- transform(dataFrame, newColumnName =所需公式)
```

因此，要利用 transform()求得两 column 之和并将结果保存为新 column，大家可以利用以下代码来实现：

```
dataFrame <- transform(dataFrame, newColumn = oldColumn1 + oldColumn2)
```

要利用 transform()向我们的数据框中添加利润率 column，大家需要这样操作：

```
companiesData <- transform(companiesData, margin = (profit/revenue) * 100)
```

接下来，我们可以利用 round()函数将新 column 中的数值调整为只取小数点后一位。或者，我们也可以采取一步到位的方法，直接创建一个仅保留小数点后一位的新 column：

```
companiesData <- transform(companiesData, margin = round((profit/revenue) * 100, 1))
```

下面我们来总结 round()函数的使用方法：大家可以通过负数形式表达“小数点后的保留位数”。举例来说，round(73842.421，1)保留的就是一位——结果为 73842.4，而 round(73842.421，-3)则代表取最接近的千位整数，也就是 74000。

语法三:R 语言的 apply()函数顾名思义，会将某个函数“应用”在数据框（或者多种其它 R 数据结构，但我们目前姑且只关注数据框这一种）当中。它的语法与前两种函数相比要复杂一些，但在某些难度较高的计算过程中会起到重要作用。

apply() 的基本格式为:

```
dataFrame$newColumn <- apply(dataFrame, 1, function(x) { . . . } )
```

以上代码行的作用是在数据框内创建一个名为“newColumn”的新 column；其中的内容将由{…}的具体代码决定。

下面我们来分别解释以上代码行中各 apply()参数的具体含义。第一项 apply()参数代表着现有数据框。第二项参数，在本示例中为“1”，意思是“在 row 中应用一项函数”。如果该参数为 2，则代表“在 column 中应用一项函数”。如果大家打算对当前 column 而非 row 进行求和或者求平均值，那么直接修改这条参数就能轻松达到目的。

第三项参数为 function(x)，很明显具体内容有待写入。具体来说，在其它情况下 function()部分将保持不变，但“x”则可以是任何变量名称。那在我们的示例中，x 代表着什么呢？它的意思是将所有条目（row 或者 column）都将由 apply()进行遍历。

最后，{…}代表我们要对遍历的每项条目进行何种操作。

请注意，apply()会对所有 row 或者 column 内的每一项条目进行查找发实现函数应用。如果大家应用的函数只能作用于数字条目、而当前数据框中某些 column 的内容并非数字，就可能引发问题。

我们的财务统计样本数据正好符合这种情况。对于数据变量来说，以下代码无法正常生效：

```
apply(companiesData, 1, function(x) sum(x))
```

为什么会这样？因为 apply()会试图将每一行中的条目进行相加，而公司名称是无法被计入公式的。

为了让 apply()只作用于数据框中的特定 column 中，例如将营收与利润中的数值相加（当然，我承认这样的算法在现实世界的财务分析工作中不太可能发生），我们需要将对应的数据框子集作为我们的第一项参数。也就是说，我们不再让 apply()作用于整套数据框，而只让它针对营收与利润两列起效，具体代码如下所示：

```
1.  apply(companiesData[,c('revenue', 'profit')], 1, function(x)
    sum(x))
```

请大家注意其中的[c('revenue', 'profit')]，这部分内容位于数据框名称之外，它的作用是强调求和操作“只作用于营收与利润两列”。

下面我们要做的是将 apply 的计算结果保存在新的 column 当中，代码如下：

```
1.  companiesData$sums <- apply(companiesData[,c('revenue', 'pro
    fit')], 1, function(x) sum(x))
```

对于单纯的求和函数来说，这样的效果已经合格了。不过让我们再想想前面提到过的例子——根据每一行中的条目计算公司利润率。在这种情况下，我们需要将利润与营收以特定顺序加以排列——也就是用利润除以营收，而不是相反——然后再乘以 100。

我们该如何利用匿名 function(x)中让 apply()以特定顺序处理多个条目？答案是将我们的匿名函数分别命名为 x[1]、x[2]并以此类推，利用它们指代不同的条目：

```
1.  companiesData$margin <- apply(companiesData[,c('revenue', 'p
    rofit')], 1, function(x) { (x[2]/x[1]) * 100 } )
```

以上代码行创建出一项利用第二条目除以第一条目的匿名函数——由于在 companiesData[,c('revenue', 'profit')]当中，营收在前而利润在后，因此第二条目就指代利润而第一条目指代营收。这种方式在我们的示例中是可地的，因为其中只涉及两项条目，即营收与利润——请记住，我们一定要让 apply()只作用于这两个对应 column。

语法四。mapply()与更简单的 sapply()也可以实现将函数应用在数据框某列中的效果——但不一定对所有列都有效——而且最重要的是，跟它们两位打交道时我们不用再考虑 x[1]和 x[2]这种麻烦事。要利用 mapply()在数据框中生成一个新 column，我们需遵循以下格式：

```
dataFrame$newColumn <- mapply(someFunction, dataFrame$column
1, dataFrame$column2, dataFrame$column3)
```

以上代码会将函数 someFunction()应用到数据框各行 column1、column2 以及 column3 中的数据身上。

请注意，mapply()的第一个参数为“函数名称”，而非公式或者等式。因此如果我们希望像前面提到的那样计算“利润除以营收再乘以 100”的结果，则需要首先在自己的函数中写入这一计算，然后再将其交给 mapply()。

下面我们看来如何创建名为 profitMargin()的函数，其中包含两个变量——在本文的示例中，我们将其称为 netIncome（净收入）与 revenue（营收）——第一个变量除以第二个变更再乘以 100，结果取小数点后一位：

```
profitMargin <- function(netIncome, revenue)
{ mar <- (netIncome/revenue) * 100
mar <- round(mar, 1)
return(mar)
}
```

现在我们可以利用这项由用户自己创建的命名函数与 mapply()协作了：

```
companiesData$margin <- mapply(profitMargin, companiesData$p
rofit, companiesData$revenue)
```

或者，我们也可以在 mapply()当中创建一项匿名函数：

```
companiesData$margin <- mapply(function(x, y) round((x/y) *
100, 1), companiesData$profit, companiesData$revenue)
```

与 transform()相比，mapply()的优势之一在于我们可以使用来自不同数据框的 column（请注意，如果各列的长度不同，则可能无法正确起效）。另一项优势是，mapply()在某函数拥有多个参数时，其将函数应用于数据 vector 时的语法更为精致，如下所示：

```
1.  mapply(someFunction, vector1, vector2, vector3)
```

sapply()与 mapply()在语法上略有不同，而且如今 R 语言中的应用类函数也相当丰富。我不打算在本文中再多加讨论，但看到这里，相信大家应该会明白为什么 R 语言大师 Hadley Wickham 会创建出一套名为 plyr 的定制软件包。很显然，其中所选取的函数都使用相当的语法，从而使 R 语言的功能更趋合理化。（我们在下一节中将谈到 plyr。）

如果大家需要进一步了解 R 语言中的各种应用类选项，我推荐各位阅读由 Neil Saunders 撰写的《R 语言中的‘apply’简介》。

原文：
http://developer.51cto.com/art/201312/423612.htm?utm_source=tuicool

独立开发者的自白：Objective-C 最糟糕的 13 件事

摘要：本文的作者是一名具有多年开发经验的独立游戏开发者，他从一个专业开发者视角深入剖析 Objective-C，将其与其他编程语言相比，解读 Objective-C 的优与缺，并总结出 Objective-C 的 13 件最为糟糕的事。

本文的作者 Anton Zherdev 是一名具有多年开发经验的独立游戏开发者，他从一个专业开发者的视角深入剖析 Objective-C，将其与 C、Java 等其他语言相比，解读 Objective-C 的优与缺，以最为精炼的话语总结出他所认为的 Objective-C 的 13 件最为糟糕的事，直指 Objective-C 的不足之处，与列位开发者分享。以下为文章全文：

1. 笨重的语法

在 Objective-C 中，必须要编写大量的代码来声明一个类或属性。对比下面的 Objective-C 和 Scala 的代码段，效果不言而喻。

[js] view plaincopy

```
@interface Foo : NSObject
@property (readonly) int bar;
- (instancetype)initWithBar:(int)bar;
+ (instancetype)fooWithBar:(int)bar;
@end
@implementation Foo
- (instancetype)initWithBar:(int)bar {
  self = [super init];
  if (self) {
    _bar = bar;
  }
  return self;
}
+ (instancetype)fooWithBar:(int)bar {
  return [[self alloc] initWithBar:bar];
}
@end
```

[js] view plaincopy

```
1.    class Foo {
2.      private int bar;
3.      public Foo(int bar) {
4.          this.bar = bar;
5.      }
6.      public int getBar() {
7.          return bar;
8.      }
9.    }
```

[js] view plaincopy

```
1.    class Foo(bar : Int)
```

2. 方括号

前缀括号是一件非常虐心的事情，尽管 Objective-C 有 IDE 自动解决这一问题，却也大大降低了代码的可读性。

3. 内存管理

与具有垃圾回收器的编程语言相比，Objective-C 的内存更容易泄漏。虽然垃圾回收器可能会导致程序不定时回收，但也可通过设置来避免。在 Objective-C 中，尽管已经有 ARC 来解决这一问题，并且，开发者可以很好地利用它来解决一些**弱引用**的问题，却还存在着无法解析引用周期的状况。

举个例子，如果你在 **block** 中使用 self，并将其发送给另一个存储该 block 的类，便会导致内存泄漏，而且还很难发现。并且，如果想要在类字段中保存 block，就必须要将其 copy 过来，否则，当 block 被调用时，程序也就崩溃了。

4. 动态且不安全的类型系统

Objective-C 有一个非常奇怪的**类型系统**。任何消息，只要在视图中可声明，就可以将其发送给 id 类对象，但却无法回避 id。

```
[js] view plaincopy
```

```
1.  @interface Foo : NSObject
2.  - (void)foo;
3.  @end
4.  @implementation Foo
5.  - (void)foo{}
6.  @end
7.
8.  @interface Bar : NSObject
9.  - (void)bar;
10. @end
11. @implementation Bar
12. - (void)bar {
13.   [(id)self foo]; //OK
14.   [(id)self bar]; //OK, but runtime error
15.   [(id)self qux]; //Compiler error
16.   Foo* foo = self; //OK
17. }
18. @end
```

5. 不支持泛型

在 Objective-C 中，要想查看容器类所属是不可能的，而且，编译器也不能进行检查。这方面，Java 显然要好得多。

```
[js] view plaincopy
```

```
1.  NSArray* array = [foo array];
2.  id item = [array objectAtIndex:0]; //item is anything
```

```
[js] view plaincopy
```

```
1.  List<A> array = foo.getArray();
2.  A item = array.get(0);
```

6. 核心库集匮乏

在 Objective-C 的核心库中，缺少诸如分类设置、字典（地图）、链表、队列等实用集。没有它们，在进行红黑树分类和字典等开发管理时会花费大量的时间。

还有一个问题就是缺乏几项非常不错的功能，尤其是函数式编程能力有所缺失，尽管如此，但在 Objective 中，也有非常不错的一项功能，就是开发者可以使用分类简单地扩展核心集。

7. 缺少枚举

尽管 Objective-C 包含有 C 的枚举，但却仅仅只是一组变量，开发者必须要编写代码来实现一些类似于 Java 的枚举，比如链接属性等。

```
[js] view plaincopy
```

```
1.  enum Foo {
2.      Foo1(1),
3.      Foo2(2),
```

```
4.        Foo2(3);
5.        final int bar;
6.        Foo(int bar) {
7.           this.bar = bar;
8.        }
9.     }
```

8. 可怕的 block 语法

block 是 Objective-C 一项非常强大的功能，但我却不知如何声明带有 block 类型的变量或函数参数。看下面这段代码便可知晓。

[js] view plaincopy

```
1.    //Declare variable foo
2.    void (^foo)(id obj, NSUInteger idx, BOOL *stop);
```

9. 操作符重载缺失

如果说无需操作符重载的话，难免有些欠妥，因为定义向量数学运算符是件很正常的事情，它使代码更具有可读性。

[js] view plaincopy

```
1.    [[[a add:b] sub:[c mul:f]] div:f];
2.    (a + b - c*f)/f;
```

10. 匿名类不足

定义一个协议或接口并不像想象中那么简单，要想轻而易举地实现，就必须要先实现一个完整的类。

[js] view plaincopy

```
1.  @protocol I
2.  - (void)f;
3.  @end
4.
5.  @interface Foo : NSObject
6.  - (void)foo;
7.  - (void)qux;
8.  @end
9.  @implementation Foo
10. - (void)foo {
11.   id<I> i = [[Baz alloc] initWithFoo:self];
12. }
13. @end
14.
15. @interface Baz : NSObject<I>
16. - (instancetype)initWithFoo:(Foo*)foo;
17. @end
18. @implementation B {
19.   Foo* _foo;
20. }
```

```
21.
22.  - (instancetype)initWithFoo:(Foo*)foo {
23.    self = [super init];
24.    if(self) {
25.    _foo = foo;
26.  }
27.    return self;
28.  }
29.
30.  - (void)f {
31.    [_foo bar];
32.  }
33.  @end
```

[js] view plaincopy

```
1.   interface I {
2.     void f();
3.   }
4.
5.   class Foo {
6.     void foo() {
7.         I i = new I() {
```

```
        void f() {
            bar();
        }
    };
  }
  void bar() {
  }
}
```

11. 糟糕的构造函数

使用构造函数创建新对象很常见，但在 Objective-C 中，要想创建对象，还需调用两个函数。当然，开发者可以编写方法直接避免该问题的发生。

[js] view plaincopy

```
@interface Foo : NSObject
- (instancetype)initWithBar:(int)bar;
+ (instancetype)newWithBar:(int)bar;
@end
@implementation Foo {
    int* _bar;
}
- (instancetype)initWithBar:(int)bar {
  self = [super init];
```

```
10.     if(self) {
11.     _bar = bar;
12.    }
13.    return self;
14.  }
15.  + (instancetype)newWithBar:(int)bar {
16.    return [[self alloc] initWithI:bar];
17.  }
18.  @end
```

12. 不透明的数值包装器

在 Objective-C 中，要想在集合或其他容器中使用数值是件很费劲的事情，除了原有代码之外，还需添加一个 **NSNumber 类**。

```
[js] view plaincopy
```

```
1.    int a = 1;
2.    int b = 2;
3.    int c = a + b;
4.    NSNumber* aWrap = @(a);
5.    NSNumber* bWrap = @(b);
6.    NSNumber* cWrap = @([aWrap intValue] + [bWrap intValue]);
```

13. 缺乏包管理系统

在 Objective-C 中，开发者必须要使用到前缀，以避免类名重合。此外，还要在头文件中对所需类进行声明，以防头文件冲突，编译失败。

原文：
http://www.csdn.net/article/2013-12-17/2817832-top-13-worst-things-about-objective-c?utm_source=tuicool

使用 Objective-C 一年后我对它的看法

我在一年前因需要将 RedPhone 项目从 Android 移植到 iOS 而首次接触 Objective-C。大约一个月前我负责的部分（后端：声音，网络，加密部分）已经完成。我们正等着外部的安全审查，同时在内部继续完成 UI 工作的过程中并未发现任何后端的 bug(言外之意开发质量高)。在 RedPhone 最终发布后，我对于这次工作中哪些做错了，基于 android 和 ios 的代码都有哪些不同等等做了讨论。今天想探讨一下去年过程中关于 Objective-C 的一些体验。

一年前我从未接触过 Objective-C.（一行代码也没看过）我是名 C#开发。现在我认为自己是名中级 Objective-C 开发：

虽然还有很多知识上的漏洞但我在现在使用这门语言已算是得心应手。比如，大家都知道不能把 nil 赋给数组和其他的集合类型，但就在上周我还不知道可以直接插入 NSNull 来实现。当然了，我只是假设大家只是不会把 nils 赋给某个集合。（意味着大部分人也不知道这个 workaround）

当我讨论语言时，我听起来会倾向于比较挑剔，所以请记住这篇文章不是指控 Objective-C 无法有效的处理问题。这篇文章是关于我使用 Objective-C 的经验，它和其他语言不同的地方，以及我遇到的问题。这是我的*观点*，因为编程领域充满了关于哪种语言更好的完全对立的观点（e.g. 动态 vs 静态类型编程语言），你可以相信，这仅仅是以一对多。一个相关的引用：

仅有两种语言：一种是大家抱怨的，另一种是没人使用的。 —— Bjarne Stroustrup

概述

Object-C 是 C 语言的超集，所以很自然的让我注意的第一件事是它哪里和 C 不同。基本上你在以 Obj-C 风格写代码时唯一做的类似 C 的事情(C-ish stuff)是使用头文件和使用前进行类型声明。

奇怪的是，Objective-C 使我想起的更多的不是 C 而是 Visual Basic 6。至少，是在我每日使用所遇到的问题上。Obj-C 和 VB6 两者都 喜欢让程序保持 运行下去(即使出错了，译者注)。这是不是个奇怪的现象？程序可能会错过打破当前状态的问题。另外，两种语言都使用引用计数来清理内存，这会随着你的深入造成更多内存泄露。永远警惕引入循环引用。最后，两者都没有泛型类型。把这

些话都直直的装入你的脑子里，否则你接下来的日子会很难过。

还有另一件 Obj-C 与 VB 的包袱，例如，在 Objective-C 中许多对象有 “core foundation”、 “new style” “NeXTSTEP”的形式，而自动引用计数技术早在两年前就已渗透到整个社区了，当然了，还有 C 的子集。所以，大多数情况下我们可以忽略这些历史包袱，但它们的出现，我并不是暗示说 Obj-C 与 VB 是相同的编程语言，它们有很大不同，首先，语法就有很大的不同。

语法

Objective-c 的语法很特别。传递一个消息（如调用个方法），你需要用方括号把接收消息的对象以及消息括起来。消息由参数名和值组成，因此不能像 list.Insert("test", 0)这样写，而是这样 [mutableArray insertObject:@"test" atIndex:0]。

当调用方法时给每个参数取个名字可以使代码非常清晰,但是不可避免的增加了代码量。总的来说这是件好事。另一方面，我认为让方括号在目标对象之前开着，而不是在目标对象之后（像 Java, C#, C++)是个错误。这使你更难确定你在一个表达式的位置，并且使一个简单的序列变成了深层嵌套。扩展一个表达式是个反复的事情，因为你必须回到行首添加 [。 当前输入] 时 XCode 会尝试猜测 [的相对位置，但是它老是搞错以致变得更糟。

幸运地是，Objective-c 有些语法糖可以让你在一般情况下摆脱嵌套。点记法 可以让你重写简单的 getter 表达式,如[a value] 重写为 a.value。setter,数组索引,字典查找也有类似的语法糖。他们确实为减少冗长的代码创造了奇迹。

Objective-c 也有容器了，这个是关于这门语言我真的很喜欢的第一件事。你想要个字典？只需要输入@{key:val,key2:val2,...}， 没有自动包装，因此你不得不在每一处添加@符号，像@{@"a":@1,@"b":@2,...}， 但是相比于人们过去不得不做的（大量地 NSNumber numberWithInt:，以 null 结束的 NSArray arrayWithObjectsfor:值，another key 值，最后不幸中的万幸 NSDictionary dictionaryWithObjects:forKeys:）这只是很小的代价。

我没有预料到 Object-C 会有匿名函数的功能,但是它确实有(它们叫做块)。块语法因为缺乏推理类型语法，所以你不得不写^(int x){return x*x;}用来替代 x=>x*x，但是确实足够简明可以被使用。

另外一个对于 Object-C 语法来说的必要点就是前缀使用 “+” 和 “-” 用于区分静态方法和实例方法(错误的...信息)。你将会很快的习惯它，虽然它的这些差别看起来有些滑稽。

类型系统

如果让我说一句我恨 Objective-C 的地方，那一定就是类型系统了。它很难表示出很多有用的类型(例如没有泛型)并且实际上它也不能检查你的工作(甚至在运行时)。虽然它在感觉上很动态，但是我更喜欢静态的强类型。

一开始，Objective-C 没有 null(但是 C 语言有，虽然你不常用)。替代的，Objective-C 有 nil,它很像 null 除了当你不小心在程序里使用它的时候它不会使程序停止。发给 nil 的消息不会抛出异常，它仅仅只返回了一个 nil 并且实际上不作任何的评估。

我是真的真的不喜欢 nil. 例如，假设你写了像构造函数的方法，它有一个叫 SuperImportantSecurityChecker 的参数。

你坚持参数不是 nil，因为对不合逻辑的不被执行进行安全检查是很糟糕的。你也可以写一个测试，故意导致安全失败，并且检查下确实是失败了。你做的很好，因为你忘记了用 SuperImportantSecurityChecker 参数的值初始化 SuperImportantSecurityChecker 了。像这样的错误会几年都不会被发现，这多亏了 nil。

另一个例子，当程序已经出错了，但是会继续运行，令人吃惊的竟然是缺少运行时类型检查。例如，如果你写 NSMutableArray* v = [NSArray array]，Objective-C 会高兴的将你的可变数组的指针指向一个不可变的数组（你以前甚至没有看到编译报警，因为 NSArray.array 返回 id，现在它返回 instancetype）。当你尝试去调用给“可变”数组添加对象的方法时，程序会崩溃，错误提示为“selector not found”。这个并不像不可期的 nil 那样糟糕，nil 会悄悄地丢弃要添加的项而不是崩溃，但是查找这些错误是很恼人的。

当在块上工作时缺少运行时检查是非常恶心的，因为语言允许你过度自由的指定输入类型。这种写法看起来多么诱人，[array enumerateObjectsUsingBlock:^(NSNumber* obj, NSUInteger index, BOOL *stop) { ... }];，根据你的期望，thatobjshould 通常是一个数字而不是 generalid，但是有时候你会搞砸它并且不得不追踪问题所在。我已经采取了在大多数块开始前加一行 strewingassert([obj isKindOfClass:[WhateverType class]])来先发制人的捕捉这些错误。

另一个值得注意的 Objective-C 会默默舍弃的是，如果你忘记了特定的编译选项。你或许已经知道不指定“- Objc"会导致分类方法在运行时不被接受，尽管不会在编译时被发现，但是你知道不包含-fobjc-arc-exceptions”会导致有异常抛出时 ARC 不能正确的进行清理吗？根据约定 强烈支持你不要捕获异常是合理的，并且速度好处显然不小，但是，苹果让他们的语言的默认执行不正确的行为着实让我受宠若惊。

基本上，我时常感觉 Objective-C 的类型系统是在挑战我。我也尝试写一个安全的 voip 应用来保护你免受你的压迫性的政府，但是我在用的语言实际上被设计成升级微不足道的错误来折中，而不是立马崩溃。我觉得，我将从不会犯低级错误，因此没有人会死，对吗？这是完美的。或者有些人受虐待。。。我也许也会检查它十多次。。。

XCode

语言的开发环境，是很影响你对该语言的感受的。开发 Objective-C 程序可以选择苹果公司的 XCode 工具，我个人比较习惯于使用 XCode。此外还可以选择 JetBrain 的 AppCode 开发环境来开发，虽然这个工具是收费的，但说实话我不是很喜欢，没用多久就丢弃了。

如果我给 XCode 评个分的话，满分 5 分我给 3 分，算是中规中矩吧。XCode 有许多细节做得很棒，比如， 这个 *fantastic* ，它会高亮显示匹配的花括号（闪烁黄色高亮），还有方法自动补全之后，有行内气泡提示，这也非常好用。不过也有许多不足之处。

奇怪的是，我主要抱怨的是项目文件的格式。我发誓，它*设计*就是为了引起合并冲突。我不知道是否 UUID 与各个条目、重复信息，或者每行多种信息相关。。。但我不记得最后一次我合并，而没有*手动修复&*(&ing 项目文件*。

有趣的是，对项目经常性的崩溃有些本末倒置。修复项目最快的方法是恢复

它，然后通过把对应的文件夹拖进项目中重建组，来修复包含源代码的组（这么做是有效果的，因为源码文件合并是正常的）。这个规律的重建操作保持你的组与文件系统同步，因此你的项目不会在像 github 一样的外部环境中看起来一团糟。多么“方便”啊。

我使用 XCode 的下一个问题是自动补全。它并不擅长这个。特别是当你在上面已经有了写了一半的代码，在你编辑时总是可以保证看到*部分*结果。这会诱使你认为基本的方法不存在，因为你正在盯着一个不包括它们的补全列表。XCode 自动补全的另一个情况是当你使用点语法时明显不好：它基本没工作过。输入一个空格，你得到很多的结果，包括读取器，但对应输入一个点，你将可能看不到任何结果。令人沮丧。

我对 XCode 的最后一个抱怨，也是值得提一下的，是有限的重构功能。在重命名变量和一些其他东西时，会受限制，而且也没什么效果。这真的很慢（我见过花费几分钟的情况），另一半情况就是 XCode 崩溃。老实说，如果你想要做重构的话，你应该仅仅安装 AppCode。要么用那个，要么习惯忍受 find+replace。

总结

Objective-C 还行。它有很多很好的语法和好用的编辑器，但是(从一些喜欢静态类型人们的角度来说)类型系统留下了许多可以改进的地方。

至少，这是我个人的体会。

原文：

http://www.linuxeden.com/html/news/20131223/146736.html?utm_source=tuicool

对比 iOS 中的四种数据存储

你是用什么方法来持久保存数据的？这是在几乎每一次关于 iOS 技术的交流或讨论都会被提到的问题，而且大家对这个问题的热情持续高涨。本文主要从概念上把“数据存储”这个问题进行剖析，并且结合各自特点和适用场景给大家提供一个选择的思路，并不详细介绍某一种方式的技术细节。

谈到数据储存，首先要明确区分两个概念，数据结构和储存方式。所谓数据结构就是数据存在的形式。除了基本的 NSDictionary、NSArray 和 NSSet 这些对象，还有更复杂的如：关系模型、对象图和属性列表多种结构。而存储方式则简单的分为两种：内存与闪存。内存存储是临时的，运行时有效的，但效率高，而闪存则是一种持久化存储，但产生 I/O 消耗，效率相对低。把内存数据转移到闪存中进行持久化的操作称成为归档。

二者结合起来才是完整的数据存储方案，我们最常谈起的那些：SQLite、CoreData、NSUserDefaults 等都是数据存储方案。当然在这些框架提供的方案

之外，我们自己也可以按照个性化需求订制方案。这些存储方案侧重不同，支持的形式和方式也各不相同，在不同的使用场景下表现也是各有优劣。但万变不离其宗，无论什么方案都可以用下图来解释。

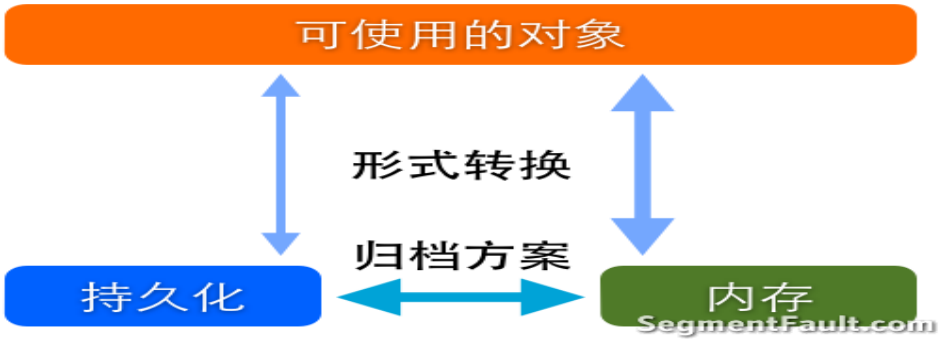


图 1，存储方案示意图

以下将对四种存储方式进行详细的介绍：

NSUserDefaults，用于存储配置信息
SQLite，用于存储查询需求较多的数据
CoreData，用于规划应用中的对象
使用基本对象类型定制的个性化缓存方案
用 NSUserDefaults 存储配置信息

NSUserDefaults 被设计用来存储设备和应用的配置信息，它通过一个工厂方法返回默认的、也是最常用到的实例对象。这个对象中储存了系统中用户的配置信息，开发者可以通过这个实例对象对这些已有的信息进行修改，也可以按照自己的需求创建新的配置项。

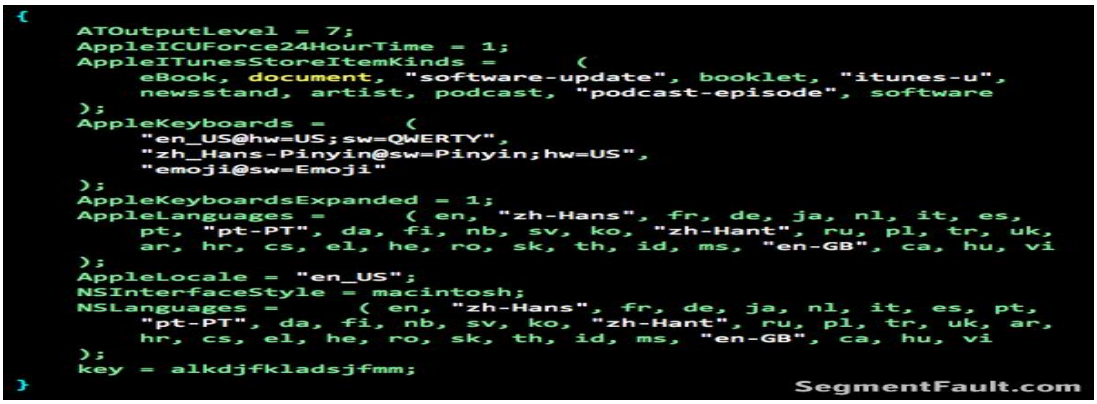


图 2，笔者手机中[NSUserDefaults standardUserDefaults]内容

NSUserDefaults 把配置信息以字典的形式组织起来，支持字典的项包括：字符串或者是数组，除此之外还支持数字等基本格式。一句话概括就是：基础类

型的小数据的字典。操作方法几乎与 NSDictionary 的操作方法无异，另外还可以通过指定返回类型的方法获取到指定类型的返回值。

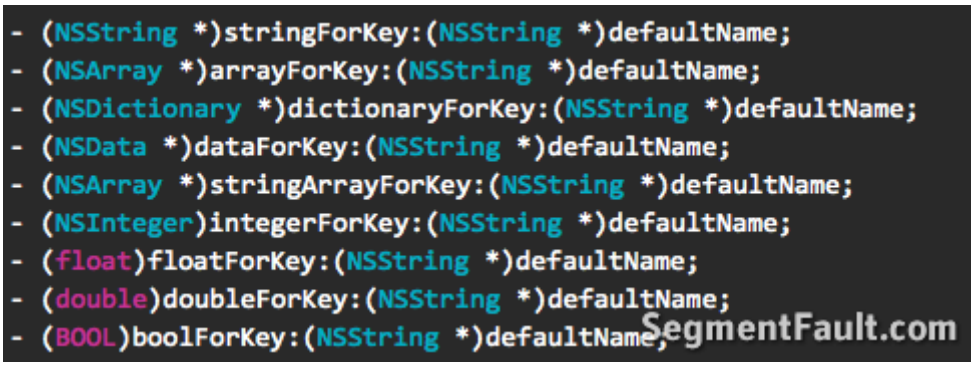


图 3，NSUserDefaults 提供的指定返回类型的方法列表

NSUserDefaults 的所有数据都放在内存里，因此操作速度很快，并还提供一个归档方法：+（void)synchronize。开发者自定义的配置项（如图 2 中的最后一项 key:alkdjfkladsjfmm）会以 plist 格式的文件归档在相应应用目录的 /Library/Preferences/[App_Bundle_Identifier].plist 文件。再次初始化获得实例对象后，框架会把用户自定义的这个配置和系统配置合并得到完整数据。

用 SQLite 存储查询需求较多的数据

iOS 的 SDK 里预置了 SQLite 的库，开发者可以自建 SQLite 数据库。SQLite 每次写入数据都会产生 IO 消耗，把数据归档到相应的文件。

SQLite 擅长处理的数据类型其实与 NSUserDefaults 差不多，也是基础类型的小数据，只是从组织形式上不同。开发者可以以关系型数据库的方式组织数据，使用 SQL DML 来管理数据。 一般来说应用中的格式化的文本类数据可以存放在数据库中，尤其是类似聊天记录、Timeline 等这些具有条件查询和排序需求的数据。

每一个数据库的句柄都会在内存中都会被分配一段缓存，用于提高查询效率。另一个方面，由于查询缓存，当产生大量句柄或数据量较大时，会出现缓存过大，造成内存浪费。

SQLite 的使用起来要比 NSUserDefaults 复杂的多，因此建议开发者使用 SQLite 要搭配一个操作控件使用，可以简化操作。笔者开发的 SQLight 是一款对 SQLite 操作的封装，把相对复杂的 SQLite 命令封装成对象和方法，可以供大家参考。大家可以在 Github 上获取这个工程的代码进一步了解。

用 CoreData 规划应用中对象

官方给出的定义是，一个支持持久化的，对象图和生命周期的自动化管理方案。严格意义上说 CoreData 是一个管理方案，他的持久化可以通过 SQLite、XML

或二进制文件储存。如官方定义所说，CoreData 的作用远远不止储存数据这么简单，它可以把整个应用中的对象建模并进行自动化的管理。

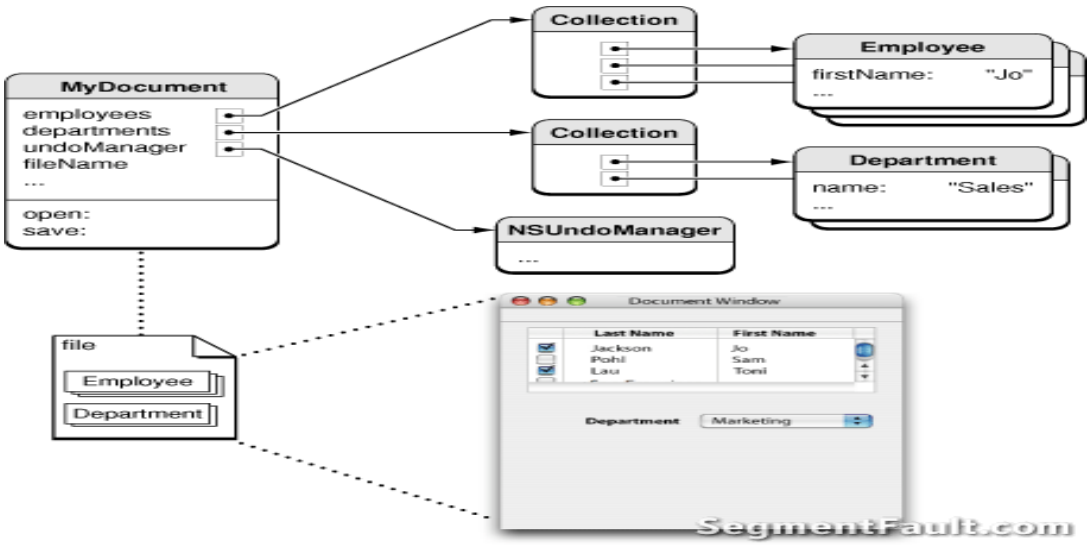


图 4，官方文档中解释 CoreData 给出的对象图示例

正如上图所示，MyDocument 是一个对象实例，有两个 Collection：Employee 和 Department，存放各自的对象列表。MyDocument、Employee 和 Department 三个对象以及他们之间的关系都通过 CoreData 建模，并可以通过 save 方法进行持久化。

从归档文件还原模型时 CoreData 并不是一次性把整个模型中的所有数据都载入内存，而是根据运行时状态，把被调用到的对象实例载入内存。框架会自动控制这个过程，从而达到控制内存消耗，避免浪费。

无论从设计原理还是使用方法上看，CoreData 都比较复杂。因此，如果仅仅是考虑缓存数据这个需求，CoreData 绝对不是一个优选方案。CoreData 的使用场景在于：整个应用使用 CoreData 规划，把应用内的数据通过 CoreData 建模，完全基于 CoreData 架构应用。

苹果官方给出的一个示例代码，结构相对简单，可以帮助大家入门 CoreData。

使用基本对象类型定制的个性化缓存方案

之前提到的 NSUserDefaults 和 SQLite 适合存储基础类型的小数据，而 CoreData 则不适合存储单一的数据，那么对于类似图片这种较大的数据要用什么方式储存呢？我给出的建议就是：自己实现一套存储方案。说到订制存储方案大家非常容易质疑，这是不是又在重新发明轮子。我可以非常明确的告诉大家，这绝不是在重新发明轮子。首先要明确，这个所谓的定制方案适用于互联网应用中对远程数据的缓存，几个限制条件缺一不可。

从需求出发分析缓存数据有哪些要求：按 Key 查找，快速读取，写入不影响正常操作，不浪费内存，支持归档。这些都是基本需求，那么再进一步或许还需要固定缓存项数量，支持

队列缓存，缓存过期等。从这些需求入手设计一个缓存方案并不十分复杂，Kache 是笔者根据开发应用的需求开发的一套缓存组件，通过分析 Kache 希望可以给大家一个思路。

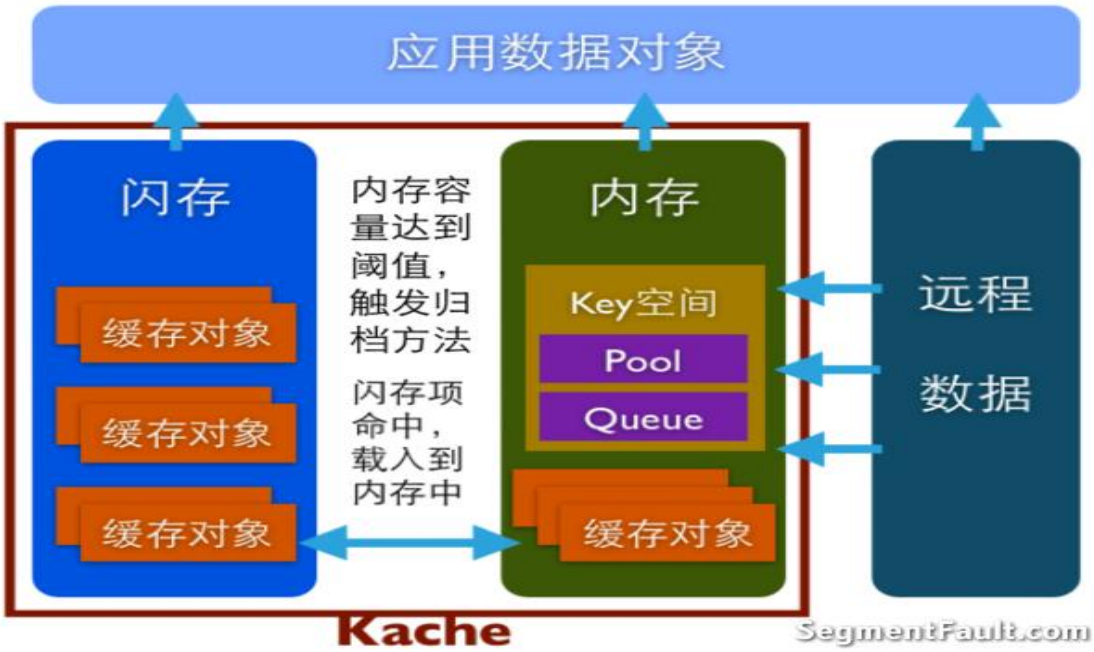


图 5，Kache 架构图

如上图所示，Kache 扮演的是一个典型缓存角色。应用加载远程数据生成应用数据对象的同时，通过 Kache 把数据缓存起来，再次请求则直接通过 Kache 获取数据。

缓存对象可以是 NSDictionary、NSArray、NSSet 或 NSData 这些可直接归档的类型，每个缓存对象对应一个 Key。缓存对象包括数据和过期时间，内存中存放在一个单例字典中，闪存中每个对象存为一个文件。Key 空间按照各种顺序存放缓存对象的 Key 集合，Pool 为固定大小的数组，当数量达到上限，最早过期的一个 Key 将被删除，对应的缓存对象也被清除。Queue 也是固定大小的数组，以先进先出的规则管理 Key 的增删。 每一次有新的缓存对象存入，自动检测 Key 空间中过期的集合并清除。

此外，控件提供 save 和 load 方法支持持久化和重新载入。

Kache 最初设计为存放图片缓存，之后也曾用于缓存文本数据，由于使用了过期和归档相结合的逻辑，可以保证大部分命中的缓存对象都在内存中，从而获取了较高的效率。读者可以从 Github 上获取 Kache 源码了解更多。

以上介绍了几种 iOS 开发中经常会遇到的储存数据方法，从其存储原理、使用方式和适用场景几方面进行进了简单的对比。事实上每一款应用都很难采用一种单一的方案完成整个应用的数据储存任务，需要根据不同的数据类型，选择最合适的方案，以便整个应用获得良好的运行时性能。

原文：

http://blog.segmentfault.com/gaosboy/1190000000363438?utm_source=tuicool

2 月 1 日起提交的应用需针对 iOS 7 优化

苹果在其开发者中心网站发布了一则新的通知，告知应用开发者以后提交审核的应用必须是使用最新版本的 Xcode 5 开发而且针对 iOS 7 系统优化过的。

从 2 月 1 日起，提交到 App Store 审核的新应用或者应用更新必须是使用最新版的 Xcode 5 开发，并且针对 iOS 7 优化过的。请查看 iOS 7 人机交互设计指导来做好准备。

这个新的规定将在明年 2 月 1 日正式施行，这也就意味着，届时如果开发者提交的应用不符合以上需求将会被拒绝。

当然这里需要说的是，苹果并没有要求开发者的应用在视觉效果上一定要 iOS 7 化，只是要求应用底层的一些东西针对 iOS 7 系统优化，必须适配 iOS 7。

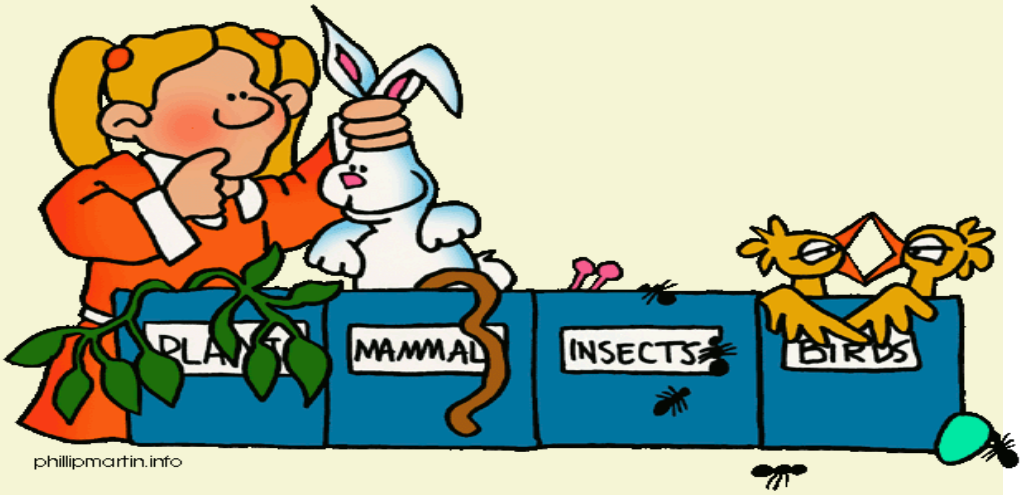
对于苹果而言，尽可能的推动软件系统大一统是他们追求的方向，这一点从他们不遗余力的推动用户设备升级到 iOS 7 系统的举动中也可以看出来。

原文：http://www.iapps.im/single/19363?utm_source=tuicool

朴素贝叶斯分类器的应用

生活中很多场合需要用到分类，比如新闻分类、病人分类等等。

本文介绍朴素贝叶斯分类器（Naive Bayes classifier），它是一种简单有效的常用分类算法。



一、病人分类的例子

让我从一个例子开始讲起，你会看到贝叶斯分类器很好懂，一点都不难。

某个医院早上收了六个门诊病人，如下表。

症状	职业	疾病
打喷嚏	护士	感冒
打喷嚏	农夫	过敏
头痛	建筑工人	脑震荡
头痛	建筑工人	感冒
打喷嚏	教师	感冒
头痛	教师	脑震荡

现在又来了第七个病人，是一个打喷嚏的建筑工人。请问他患上感冒的概率有多大？

根据贝叶斯定理：

```
P(A|B) = P(B|A) P(A) / P(B)
```

可得

```
    P(感冒|打喷嚏 x 建筑工人)
  = P(打喷嚏 x 建筑工人|感冒) x P(感冒)
  / P(打喷嚏 x 建筑工人)
```

假定"打喷嚏"和"建筑工人"这两个特征是独立的，因此，上面的等式就变成了

```
    P(感冒|打喷嚏 x 建筑工人)
  = P(打喷嚏|感冒) x P(建筑工人|感冒) x P(感冒)
  / P(打喷嚏) x P(建筑工人)
```

这是可以计算的。

```
P(感冒|打喷嚏 x 建筑工人)
= 0.66 x 0.33 x 0.5 / 0.5 x 0.33
= 0.66
```

因此，这个打喷嚏的建筑工人，有66%的概率是得了感冒。同理，可以计算这个病人患上过敏或脑震荡的概率。比较这几个概率，就可以知道他最可能得什么病。

这就是贝叶斯分类器的基本方法：在统计资料的基础上，依据某些特征，计算各个类别的概率，从而实现分类。

二、朴素贝叶斯分类器的公式

假设某个体有n项特征（Feature），分别为F_1、F_2、...、F_n。现有m个类别（Category），分别为C_1、C_2、...、C_m。贝叶斯分类器就是计算出概率最大的那个分类，也就是求下面这个算式的最大值：

```
  P(C|F1F2...Fn)
= P(F1F2...Fn|C)P(C) / P(F1F2...Fn)
```

由于 P(F1F2...Fn) 对于所有的类别都是相同的，可以省略，问题就变成了求

```
P(F1F2...Fn|C)P(C)
```

的最大值。

朴素贝叶斯分类器则是更进一步，假设所有特征都彼此独立，因此

```
  P(F1F2...Fn|C)P(C)
= P(F1|C)P(F2|C) ... P(Fn|C)P(C)
```

上式等号右边的每一项，都可以从统计资料中得到，由此就可以计算出每个类别对应的概率，从而找出最大概率的那个类。

虽然"所有特征彼此独立"这个假设，在现实中不太可能成立，但是它可以大大简化计算，而且有研究表明对分类结果的准确性影响不大。

下面再通过两个例子，来看如何使用朴素贝叶斯分类器。

三、账号分类的例子

本例摘自张洋的《算法杂货铺----分类算法之朴素贝叶斯分类》。

根据某社区网站的抽样统计，该站 10000 个账号中有 89%为真实账号（设为 C_0），11%为虚假账号（设为 C_1）。

```
C0 = 0.89

C1 = 0.11
```

接下来，就要用统计资料判断一个账号的真实性。假定某一个账号有以下三个特征：

```
F1: 日志数量/注册天数
F2: 好友数量/注册天数
F3: 是否使用真实头像（真实头像为 1，非真实头像为 0）
```

```
        F1 = 0.1
    F2 = 0.2
    F3 = 0
```

请问该账号是真实账号还是虚假账号？

方法是使用朴素贝叶斯分类器，计算下面这个计算式的值。

```
        P(F1|C)P(F2|C)P(F3|C)P(C)
```

虽然上面这些值可以从统计资料得到，但是这里有一个问题：F1 和 F2 是连续变量，不适宜按照某个特定值计算概率。

一个技巧是将连续值变为离散值，计算区间的概率。比如将 F1 分解成[0, 0.05]、(0.05, 0.2)、[0.2, +∞]三个区间，然后计算每个区间的概率。在我们这个例子中，F1 等于 0.1，落在第二个区间，所以计算的时候，就使用第二个区间的发生概率。

根据统计资料，可得：

```
    P(F1|C0) = 0.5, P(F1|C1) = 0.1
P(F2|C0) = 0.7, P(F2|C1) = 0.2
P(F3|C0) = 0.2, P(F3|C1) = 0.9
```

因此，

```
    P(F1|C0) P(F2|C0) P(F3|C0) P(C0)
    = 0.5 x 0.7 x 0.2 x 0.89
    = 0.0623

    P(F1|C1) P(F2|C1) P(F3|C1) P(C1)
    = 0.1 x 0.2 x 0.9 x 0.11
    = 0.00198
```

可以看到，虽然这个用户没有使用真实头像，但是他是真实账号的概率，比虚假账号高出 30 多倍，因此判断这个账号为真。

四、性别分类的例子

本例摘自维基百科，关于处理连续变量的另一种方法。

下面是一组人类身体特征的统计资料。

性别	身高（英尺）	体重（磅）	脚掌（英寸）
男	6	180	12
男	5.92	190	11
男	5.58	170	12
男	5.92	165	10
女	5	100	6
女	5.5	150	8
女	5.42	130	7
女	5.75	150	9

已知某人身高 6 英尺、体重 130 磅，脚掌 8 英寸，请问该人是男是女？

根据朴素贝叶斯分类器，计算下面这个式子的值。

```
P(身高|性别) x P(体重|性别) x P(脚掌|性别) x P(性别)
```

这里的困难在于，由于身高、体重、脚掌都是连续变量，不能采用离散变量的方法计算概率。而且由于样本太少，所以也无法分成区间计算。怎么办？

这时，可以假设男性和女性的身高、体重、脚掌都是正态分布，通过样本计算出均值和方差，也就是得到正态分布的密度函数。有了密度函数，就可以把值代入，算出某一点的密度函数的值。

比如，男性的身高是均值 5.855、方差 0.035 的正态分布。所以，男性的身高为 6 英尺的概率等于 1.5789（大于 1 并没有关系，因为这里是密度函数的值）。

$$p(\text{height}|\text{male}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(6-\mu)^2}{2\sigma^2}\right) \approx 1.5789$$

有了这些数据以后，就可以计算性别的分类了。

```
P(身高=6|男) x P(体重=130|男) x P(脚掌=8|男) x P(男)
= 6.1984 x e^-9

P(身高=6|女) x P(体重=130|女) x P(脚掌=8|女) x P(女)
= 5.3778 x e^-4
```

可以看到，女性的概率比男性要高出将近 10000 倍，所以判断该人为女性。

原文：
http://www.ruanyifeng.com/blog/2013/12/naive_bayes_classifier.html?utm_source=tuicool

SVN 有任何胜过 git 的地方吗？

好的技术问题通常会引出技术专家们依据经验得出的深层次的观点。但对于这样的问题的答案也很容易演变成完全基于个人喜好的情绪倾泄，而不是根据事实、标准和具体的专业知识。就比如本文的这个标题，如果你是一个 SVN 的坚定支持者，你完全可以把这句话反过来问。

我使用 SVN 有 5 年的历史了，而且现在在公司里仍然是使用 SVN。但是大概在 3 年前，我的所有个人项目都已经迁移到了 git（gitHub）上。我能想出很多 git 优于 subversion 的地方，大部分是体现在分布式优于集中式的特征上，但如果你让我说出任何 SVN 分过来胜过 git 的地方，我竟一时想不出来一个。但这就能说明 git 完胜 SVN 吗？

事实当然不是这样，就像是 Windows 和 Linux，你不能说这个一定就比那个好。最近在 stackexchange 的讨论让我学习了不少。先举个简单的例子证明有些地方你只能用 SVN 而不能用 git。谷歌的搜索排名算法，就不能放到分布式开放的代码库了。这种情况下 SVN 的集中式管理就是不二选择。下面就来条理的看看 Subversion 在哪些环境下比 git 更适用。

Subversion 是集中式管理的数据仓库

虽然速度快和多副本等 git 分布式数据仓库显而易见的好处吸引了很多人的喜爱，但在很多情况下，一个集中式的数据仓库却是更合适的。例如，如果你

有一些核心代码想只允许部分人能访问，把它放到 git 里必然是你不希望的。很多的企业都是将它们的代码集中管理的，我猜，所有(重要)政府项目估计都使用的是集中式数据仓库的版本控制系统。

Subversion 的理念符合常规思维

这是说，很多人(特别是管理者或老板)对版本号有一种习惯的认识，把开发视作一种按时间的线性发展轨迹，这在他们脑子里根深蒂固。并不是找借口，Git 的随意性并不是很容易去理解，你也许注意到了，任何一本关于 Git 的书都会在第一章第一节告诉你要抛弃脑子里所有的传统观念，重新认识。

Subversion 只提供一种途径，没有第二选择

SVN 是一个版本控制系统，它只提供**一种**方式做这些，每个人都使用相同的方法。就是这样。这使得你将代码从 SVN 迁移到其它集中式管理的 VCS 或从其它集中式管理的 VCS 迁进来变得很容易。Git 并不仅仅是一个版本控制系统——它实际上是一个文件系统，它里面有很多的拓扑学知识来支持你如何在不同的环境中架设代码仓库——并且没有一个统一的标准。选择一个合适的拓扑结构就成了难题。

其它一些优势：

- SVN 支持空目录
- SVN 有更好的 Windows 平台支持
- SVN 可以 check out/clone 一个子树(sub-tree)
- SVN 支持特权访问控制 svn lock，在处理很难合并的文件时非常有用
- SVN 支持二进制文件，更容易处理大文件(不需要把老版本拷来拷去)
- 提交文件相对简单，因为没有 pull/push 操作，本地修改通过 svn update 自动的执行了同步代码的功能。

原文：
http://www.aqee.net/what-does-svn-do-better-than-git/?utm_source=tuicool

单元化与分布式架构的切分问题

单元化是将一个系统的架构按某种数据特征维度进行垂直的划分，比如网站有 100 万用户，如按照用户维度进行划分，则可以分成 10 个单元，每个单元存储 10 万用户资料。单元化的一些收益如下

- 由于每个单元数据规模可控，相关维度内的所有资料可放在一个数据库中（如上例中的用户资料），不需要复杂的 sharding 分库分表逻辑，存储及缓存访问得到极大的简化。同时开发也变得简单，工程师不需要有丰富的“大规模大并发系统”开发经验。
- 同时由于计算离存储更近，也可以让数据离用户更近，比如用户数据存储在地理上靠近用户的位置，数据有了更好的局部性（locality），因此也会获得更好的访问性能。部署上相关单元的前端、缓存、数据库、数据挖掘等节点可在同一个机柜，架构上让大数据的访问变得低廉，也在部分程度上让大数据更为快速及敏捷。
- 可以自然支持不同用户分片支持不同的功能特性，天然的 A/B testing 试验场。

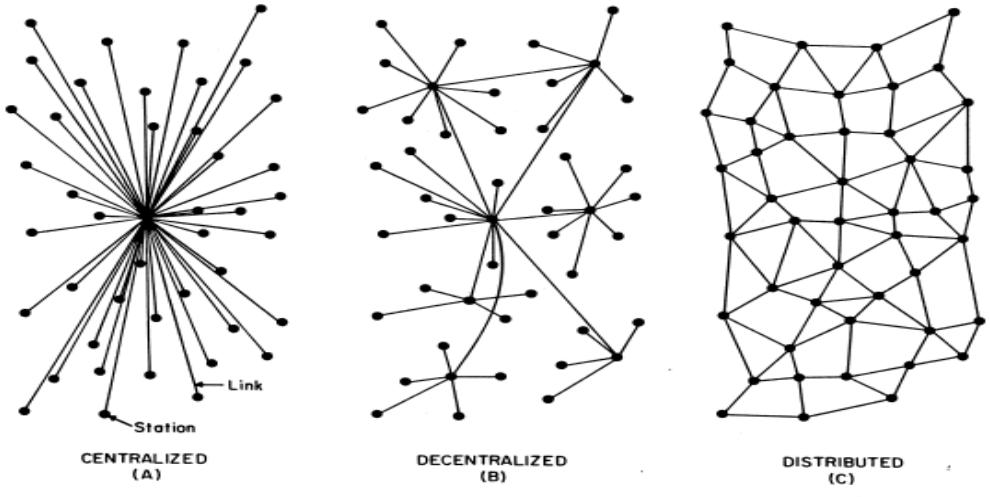


FIG. 1 — Centralized, Decentralized and Distributed Networks

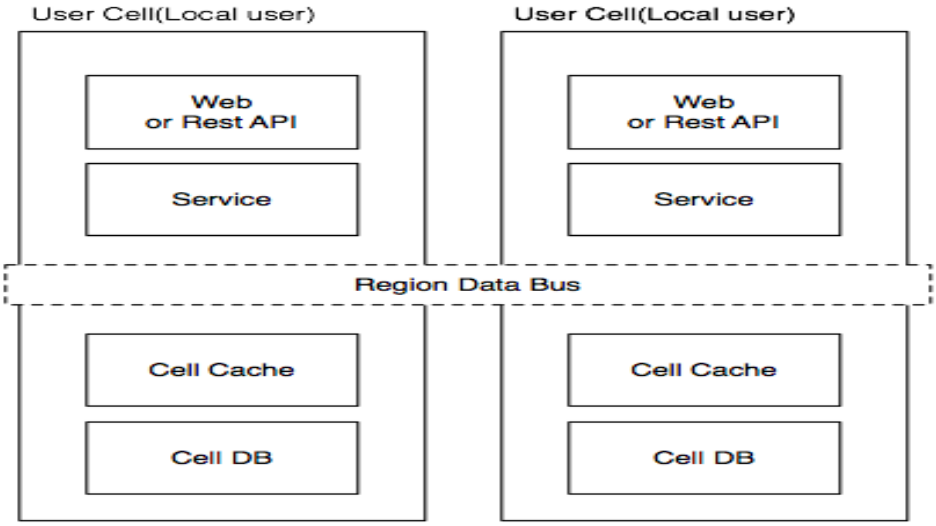
分布式是将一个系统的数据分布到多个单元，以便使系统能够 scale out，具有更好的可扩展性。当今大型网站基本上是分布式设计的。分布式系统除了机房内的，考虑到系统扩展性、用户访问的便捷性、机房规模的物理限制、异地容灾（比如 2013 年 4 月及 8 月的微信机房故障）等需要，大型系统也会考虑地理分布在多个机房。

在社交网络中，由于数据的网状访问，单元化会碰到较难选择合适的单元化切分维度的问题，比如按用户或按内容进行单元化不能很好的适应数据访问局部性的问题，同时地理分布式也面临相似问题，由于社交网络中用户的页面需要访问的，所有地理分布的机房都同步需要全量数据，导致部署和维护成本较高。

假定系统中存在一个跨单元的数据访问总线，并且总线的访问满足

1. 局部性，单元内的访问，大部分的数据可以在单元内命中。
2. 封装性，单元内的应用程序使用统一的方法访问数据，不需要关注数据的具体位置。

那么这个数据总线应该如何设计？放在哪个层级比较合适？比如 service layer, cache or storage(e.g. Google Spanner)?



Similar Posts:

- 分布式 key/value store 演讲草稿(一)
- Yahoo!的分布式数据平台 PNUTS 简介及感悟
- 构建可扩展的微博架构(qcon beijing 2010 演讲)
- 基于分发机制的公众订阅平台设计
- 广州技术沙龙安排

原文：http://www.udpwork.com/item/11325.html?utm_source=tuicool

高性能、高流量 Java Web 站点打造的 22 条建议

从 2005 年-2013 年，Ashwanth Fernando 曾供职于 Best Buy、Pearson VUE、Walgreens、Walmart eCommerce 等多家知名公司，现在 Apple 从事高级工程师、平台工程师一职，拥有丰富的高流量 Web 应用程序打造及架构经验，近日 Ashwanth 撰文分享了他的高流量 Web 软件打造经验。

下为译文

受 Joshua Bloch 写的《Effective Java》启发，我想分享自己关于建立高流量 Web 软件的整体建议。这些术语中的一些可能不仅仅关于软件设计也关于工程组织、文化等相关领域。

1. 考虑使用不止一个数据中心

在商务领域，一直存在许多恐怖的道听途说，而这些恐慌都因为他们只使用了单一的数据中心。如果你想在自然灾害或者电力供应故障中幸免，那么请使用多于 1 个的数据中心，使用 active-active 模式来配置你所有的数据中心。虽然在开销上可能会有所增加，但是比只使用单 active 的配置要值得多——因为在 passive 和 active 副本上，总会发现有些数据片不一致。

2. 考虑使用稀疏数据中心部署

不管是通过 PaaS，还是运营团队进行，当软件集群被部署到同一个数据中心的机架上时，确保这些机架使用不同的电力供应。你不可能保证机架供电的万无一失，一旦失败将会导致整个机架上服务器的丢失，这个时候你绝对不会希望整个数据中心都只连在一个电路上。

3. 考虑使用私有云来组织资源

IaaS 开源解决方案 Openstack 等其他的软件至今尚未成熟，需要庞大的团队来运营，在运行期间会产生各种各样的问题，除非你有足够的预算，否则别考虑建立一个私有的云服务。然而，私有云可以提供众多优势。首先在部署方面就可以进行众多的定制化，这远比 AWS 或者是 Rackspace 货架上的选择要多。其次它允

许你做许多的硬件定制化，就好比在硬件层次的 Oracle 就比准虚拟化环境快得多。

4. 考虑使用 PaaS 做解决方案

为软件释放投入巨量人力进行部署的日子已接近尽头，各个机构在敏捷及快速市场投放上绞尽脑汁，而 PaaS 无疑会加速这个部署过程。它允许特性尽可能快的发布，同时也能让开发者得到极大的满足。这是个非常好的开始，给予开发者部署集维护自己软件的工具，这将给工作积极性带来很大的提高。同时，越来越多的开发者甚至不愿意加入没有自动化软件部署系统的公司。更少的领导，更简化的环节，将给你带来无与伦比的效率。

5. 如果使用 Oracle 或者 MySQL，只做基于主键的查询

只有在 RAC 中存在很少的 Artifacts 时，Oracle 才能在流量高峰时获得最佳性能。尽可能避免使用 Referential Integrity、Triggers、Materialized Views、Views、Stored Procedures 和其他的 Oracle Artifacts。Triggers 可以在从数据访问层实现。Stored Procedures 可以完全转移到应用层。数据库只用来存储数据，基于字段进行存储而不是主键，使用类似 Lucene 的索引器做表的索引，使用一个允许在结果集上做基于其他字段的查询，这将会返回这个记录的主键，而这个主关键字可以进一步被用来拿取记录。<="" p="">

6. 考虑使用 Oracle 或者 MySQL 分片

当 schema 达到临界点，Oracle 的可伸缩性将被限制，这里建议你对 schema 做基于功能（比如订单，产品目录，促销活动，客户等）上的分片，同时也为高密度表做 key shards。为 key shards 使用一致性哈希，这样当一个新的 RAC 被添加 RAC 集时，你不再需要遍历所有 RAC 中的键，以获悉哪些键需要被移动到键的分片中。

7. 如果你使用 Oracle 做 RDBMS，考虑使用 Data Guard 及 Golden Gate

使用这两种技术将大大简化甲骨文的运营周期，Data Guard 允许一个近实时 passive 读副本（没有客户端会与之连接），而 Golden Gate 则允许一个近实时的 active 读写副本。

推荐的部署拓扑之一就是为同个数据中心的每个分片配置 1 个 Data Guard；使用 Golden Gate 来备份其他数据中心的每一个分片。

注意：Golden Gate 只是近实时

8. 为 Oracle 或者 MySQL 添加数据访问层

假设你有一个可以接受 500 个连接的 Oracle RAC，而你有 25 个 jBoss 实例和这个甲骨文 RAC 对话，每个 Jboss 实例配置范围 10 到 50 的数据库连接池。

当 jBoss 集群开启时，连接到 Oracle 的数目为 250（25 乘 10），一切运行良好。随着流量快到 jBoss 集群的峰值，想象一下将会发生什么。在某个点后，Oracle 将开始拒绝连接。

因此建议通过一个 Multiplexer 层建立一个 Multiplexe 应用程序服务器连接。可以是一个简单的 netty 应用，这个应用运行在一个每个 netty 节点仅能够与 Oracle 建立 25 个连接的集群上，但是对入站连接来者不拒。它会将所有的连接循环传递给 Oracle，但是绝对不会超过 25 个，同时还使用 Oracle JDBC 驱动与 Oracle 通信。

9. 避免跨数据中心事务

当下，这已经是非常简单的事情，但是在任何地方都非常适用，包括 Oracle。在两个数据不同数据中心，不要适用 1 个 XA 适配器去做跨数据中心事务，这将导致相当长时间的应用线程阻塞，直到两个阶段的提交完成，因此将带来你的应用程序服务、服务和所有同步上传流崩溃，最终会因为线程数量增加而导致整个应用程序崩溃，比如在类似 Black Friday 流量情况下。

10. 考虑分布式缓存框架

Memcached、Counbase 是最常用的选择。但实际上，卸载非易失性数据到一个中心缓存集群上，确实没必要在每个 JVM 上做相同的拷贝。但是确实需要设置小数量的 JVM 堆作为分布式缓存的一个 MRU 缓存，这样的话，缓存集群本身将会受到非常少的网络调用。

- 在 JVM 上大多数分布式缓存支持本地缓存的概念，它将储存最常用的对象。
- JVM 上，GC 的 pause time 同样被最小化了，因为对象图中需要遍历的对象比以前更少了。
- Warmup 过程是必不可少的，这可以帮助将数据导入分布式缓存，这个过程应该在晚上或者是用户访问量低的时候。

原文：
http://www.csdn.net/article/2013-12-20/2817861-22-recommendations-for-building-effective-high-traffic-web-application?utm_source=tuicool

洪强宁介绍 Douban App Engine 的架构与特性

豆瓣网首席架构师洪强宁最近在 PyCon China 分享了他们研发两年的成果：Douban App Engine（DAE）。InfoQ 中文站编辑在现场对洪强宁的分享（PPT 下载）进行了记录，提取重点内容如下：

DAE 是专门针对 Python 做的私有 PaaS 平台，目前已经支撑了 427 个应用，其中 126 个是对外应用。126 个对外应用包括：

- 豆瓣电台的 Bubbler
- 豆瓣移动版网站
- 豆瓣 FM
- 豆瓣电影
- 豆瓣小组

对内应用则包括 Douban Code 平台、豆瓣上线系统 Up 平台、用于人事管理和活动组织的豆瓣花名册、验证深度学习算法有效性的豆瓣电影评论数据分析系统精灵宝钻等等。

DAE 目前每天处理 2.4 亿动态请求，峰值可达到 5K qps，运行在 32 台服务器上（豆瓣称之为 32 个节点）。

开发 DAE 的原因

洪强宁介绍开发 DAE 的原因，包括几个方面：

1. 用 Git 替换 SVN。此前，豆瓣所有的代码都在一个 SVN repo 上，commit 数量将近 17 万。PyCon 北京之前，豆瓣正式停用 SVN，将所有代码转移到 4.6k 个 git repo 上，目前有 2.8K 个 fork。项目数量以每一两天一个新项目的速度增长
2. 基础设施公用，不用给每一个应用都配一套 MySQL、BeansDB、Memcache、MQ 什么的了
3. 大大简化了新项目启动的操作：你只需要 dae create 即可创建你的项目，dae serve 即可测试，dae deploy 即可上线
4. 最佳实践可以自动实施到所有的项目上，这包括每次提交都进入持续集成系统进行自动化测试，分级上线，状态收集和故障报告的标准化等

基于上述好处，DAE 的存在还带来了另一个极大的好处：运维工作的可扩展性。如果没有 DAE，那么 SA 的工作量跟服务器数量、应用种类的数量是呈平方增长关系的。有了 DAE，只需要 4 个 SA 就能完成所有的豆瓣运维。

DAE 的架构设计

DAE 的架构是这样设计的：

每一个应用都有一个 app.yaml 定义这个应用的版本、是 sync 模式还是异步模式、出了问题该通知谁等信息。

应用依赖采用类 pip 的方案解决，只需执行 dae install package 即可安装该 package 以及相关的所有依赖。豆瓣提供了一个 pypi 的镜像供大家使用。

实例分为三种：web、service 和 daemon。

一个 web 实例就是一个 gunicorn。使用 gunicorn 的原因是因为它是用 Python 写的，适合豆瓣对它做二次开发，而且 gunicorn 还有一些很好的特性，比如 graceful restart。

service 是用于在应用和应用之间传递信息的，用 gunicorn 配合 thrift 实现。

daemon 是那种长期运行、不会死掉的守护进程，采用 ZooKeeper 做全局管理。有了这个就可以实现一些其他的应用，可以保证 Message Queue 里有固定数量的消费者，通过定时产生一些消息，让队列的消费者执行特定程序来实现定时任务。

路由系统采用两级 Nginx 结构，请求过来之后判断它是找哪个应用的哪个实例，给它做一个 HTTP proxy 过去。基本上，长期没人访问的应用消耗是很小的，仅占用硬盘空间。这里使用了 unix socket 而避免使用 tcp，是看中了 unix socket 的文件属性，实现更简单。

所有的基础服务，如 MySQL、memcache、doubandb、cdn、mail、irc 等，都是通过 API 的方式提供。业务访问不同的服务会有不同的前缀，通过这个实现隔离和监控。

DAE 的特点

根据洪强宁的介绍，DAE 本身的资源占用是很低的，因为是用进程来做资源分配，用 Unix 账号做资源隔离，监控则交给外部的监控系统。

另外，DAE 是一个自己实现自己的系统：通过一个 DAE 应用部署应用，通过一个 DAE 应用管理应用，一个 DAE 应用 scale 所有的应用，而 DAE 的代码也是托管在一个 DAE 应用上，会有循环依赖。

DAE 自身的升级是通过开发集群->beta 集群->stable 集群逐级上线的，目的是在发布到外部应用之前 DAE 系统已经经过了内部系统的测试。

DAE 在部署应用时，会分批逐步在各节点部署，部署后会自动测试，测试不通过就会自动回滚，这样来保证应用部署过程中和部署后的可用性。

另外,DAE 的环境是不受限的,比如 C 扩展,比如 Python 的 fork、subprocess、multiprocessing 等在其他 PaaS 上禁止使用的特性，在 DAE 上都是允许的。这样当然会造成一些潜在的问题，比如你的应用 fork 出来一堆线程没有处理，就会在那里占用很多资源。所以我们做了一个曹娥来解决这个问题，在父进程死掉的时候杀死所有 fork 出来的子进程。
接下来希望做的一些事情：

- 让 DAE 成为豆瓣所有应用的平台
- 采用 cgroups 实施资源限制，以免外部监控跟不上的情况
- 开发新的服务系统，解决在应用 thrift 中发现的一些问题，和 thrift 互为补充
- 自动实现跨 IDC
- 实现 QoS，让任何应用的问题不会影响其他应用的表现

DAE 并没有计划做公有云服务，因为原本的设计采用的 Unix 账号模式，设计目标是数千个 app 的容量，很明显是无法支持公有云的用量的。

开源方面，其实一直有这个计划，会在基础库逐步开源之后再开放出来。

原文：
http://www.infoq.com/cn/news/2013/12/douban-app-engine?utm_source=tuicool

大众点评的大数据实践

摘要：大众点评网从 2011 年中开始使用 Hadoop，并专门建立团队。Hadoop 主分析集群共有 60 多个节点、700TB 的容量，月运行 30 多万个 Hadoop Job，还有 2 个 HBase 线上集群。作者将讲述这各个阶段的技术选择及改进之路。

2011 年小规模试水

这一阶段的主要工作是建立了一个小的集群，并导入了少量用户进行测试。为了满足用户的需求，我们还调研了任务调度系统和数据交换系统。

我们使用的版本是当时最新的稳定版，Hadoop 0.20.203 和 Hive 0.7.1。此后经历过多次升级与 Bugfix。现在使用的是 Hadoop 1.0.3+自有 Patch 与 Hive 0.9+自有 Patch。考虑到人手不足及自己的 Patch 不多等问题，我们采取的策略是，以 Apache 的稳定版本为基础，尽量将自己的修改提交到社区，并且应用这些还没有被接受的 Patch。因为现在 Hadoop 生态圈中还没有出现一个类似 Red Hat 地位的公司，我们也不希望被锁定在某个特定的发行版上，更重要的是 Apache Jira 与 Maillist 依然是获取 Hadoop 相关知识、解决 Hadoop 相关问题

最好的地方（Cloudera 为 CDH 建立了私有的 Jira，但人气不足），所以没有采用 Cloudera 或者 Hortonworks 的发行版。目前我们正对 Hadoop 2.1.0 进行测试。

在前期，我们团队的主要工作是 ops+solution，现在 DBA 已接手了很大一部分 ops 的工作，我们正在转向 solution+dev 的工作。

我们使用 Puppet 管理整个集群，用 Ganglia 和 Zabbix 做监控与报警。

集群搭建好，用户便开始使用，面临的第一个问题是需要任务级别的调度、报警和工作流服务。当用户的任务出现异常或其他情况时，需要以邮件或者短信的方式通知用户。而且用户的任务间可能有复杂的依赖关系，需要工作流系统来描述任务间的依赖关系。我们首先将目光投向开源项目 Apache Oozie。Oozie 是 Apache 开发的工作流引擎，以 XML 的方式描述任务及任务间的依赖，功能强大。但在测试后，发现 Oozie 并不是一个很好的选择。

Oozie 采用 XML 作为任务的配置，特别是对于 MapReduce Job，需要在 XML 里配置 Map、Reduce 类、输入输出路径、Distributed Cache 和各种参数。在运行时，先由 Oozie 提交一个 Map only 的 Job，在这个 Job 的 Map 里，再拼装用户的 Job，通过 JobClient 提交给 JobTracker。相对于 Java 编写的 Job Runner，这种 XML 的方式缺乏灵活性，而且难以调试和维 护。先提交一个 Job，再由这个 Job 提交真正 Job 的设计，我个人认为相当不优雅。

另一个问题在于，公司内的很多用户，希望调度系统不仅可以调度 Hadoop 任务，也可以调度单机任务，甚至 Spring 容器里的任务，而 Oozie 并不支持 Hadoop 集群之外的任务。

所以我们转而自行开发调度系统 Taurus（https://github.com/dianping/taurus）。Taurus 是一个调度系统， 通过时间依赖与任务依赖，触发任务的执行，并通过任务间的依赖管理将任务组织成工作流；支持 Hadoop/Hive Job、Spring 容器里的任务及一般性任务的调度/监控。

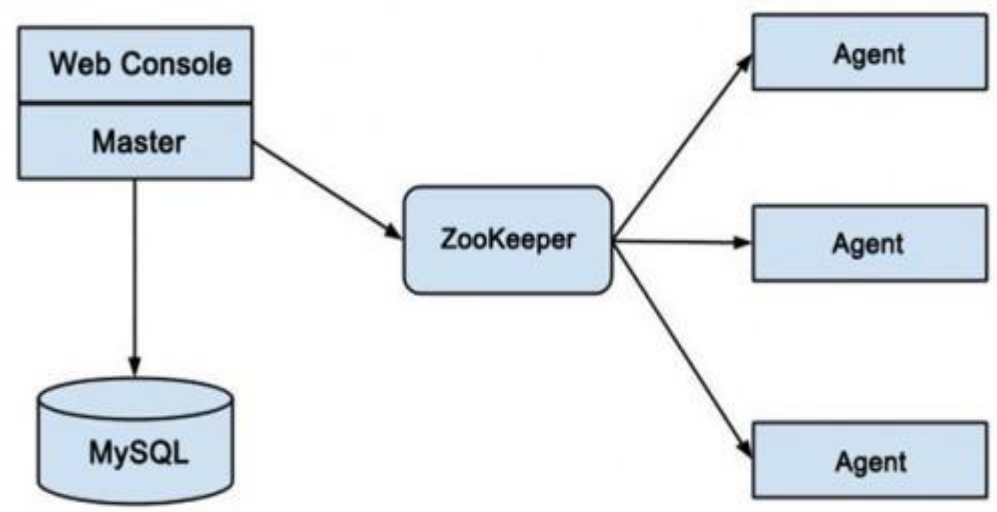


图 1 Taurus 的结构图

图 1 是 Taurus 的结构图，Taurus 的主节点称为 Master，Web 界面与 Master 在一起。用户在 Web 界面上创建任务后，写入 MySQL 做持久化存储，当 Master 判断任务触发的条件满足时，则从 MySQL 中读出 任务信息，写入 ZooKeeper；Agent 部署在用户的机器上，观察 ZooKeeper 上的变化，获得任务信息，启动任务。Taurus 在 2012 年 中上线。

另一个迫切需求是数据交换系统。用户需要将 MySQL、MongoDB 甚至文件中的数据导入到 HDFS 上进行分析。另外一些用户要将 HDFS 中生成的数据再导入 MySQL 作为报表展现或者供在线系统使用。

我们首先调研了 Apache Sqoop，它主要用于 HDFS 与关系型数据库间的数据传输。经过测试，发现 Sqoop 的主要问题在于数据的一致性。Sqoop 采用 MapReduce Job 进行数据库的插入，而 Hadoop 自带 Task 的重试机制，当一个 Task 失败，会自动重启这个 Task。这是一个很好的特性，大大提高了 Hadoop 的容错能力，但对于数据库插入操作，却带来了麻烦。

考虑有 10 个 Map，每个 Map 插入十分之一的数据，如果有一个 Map 插入到一半时 failed，再通过 Task rerun 执行成功，那么 fail 那次插入的一半数据就重复了，这在很多应用场景下是不可接受的。 而且 Sqoop 不支持 MongoDB 和 MySQL 之间的数据交换，但公司内却有这需求。最终我们参考淘宝的 DataX，于 2011 年底开始设计并开发了 Wormhole。之所以采用自行开发而没有直接使用 DataX 主要出于维护上的考虑，而且 DataX 并未形成良好的社区。

2012 年大规模应用

2012 年,出于成本、稳定性与源码级别维护性的考虑,公司的 Data Warehouse 系统由商业的 OLAP 数据库转向 Hadoop/Hive。2012 年初，Wormhole 开发完成；之后 Taurus 也上线部署；大量应用接入到 Hadoop 平台上。为了保证数据的安全性，我们开启了 Hadoop 的 Security 特性。为了提高数据的压缩率，我们将默认存储格式替换为 RCFile，并开发了 Hive Web 供公司内部使用。2012 年底，我们开始调研 HBase。

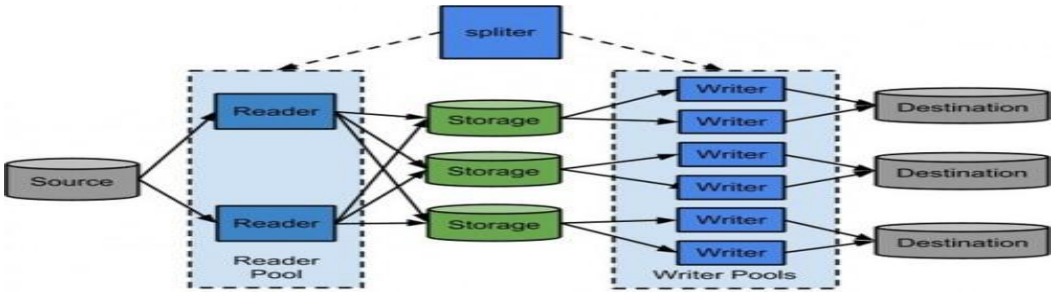


图 2 Wormhole 的结构图

Wormhole（https://github.com /dianping/wormhole）是一个结构化数据传输工具，用于解决多种异构数据源间的数据交换，具有高效、易扩展等特点，由 Reader、 Storage、Writer 三部分组成（如图 2 所示）。Reader 是个线程池，可以启动多个 Reader 线程从数据源读出数据，写入 Storage。 Writer 也是线程池，多线程的 Writer 不仅用于提高吞吐量，还用于写入多个目的地。Storage 是个双缓冲队列，如果使用一读多写，则每个目的地都拥有自己的 Storage。

当写入过程出错时，将自动执行用户配置的 Rollback 方法，消除错误状态，从而保证数据的完整性。通过开发不同的 Reader 和 Writer 插件，如 MySQL、MongoDB、Hive、HDFS、SFTP 和 Salesforce，我们就可以支持多种数据源间的数据交换。Wormhole 在大众点评内部得到了大量使用，获得了广泛好评。

随着越来越多的部门接入 Hadoop，特别是数据仓库（DW）部门接入后，我们对数据的安全性需求变得更为迫切。而 Hadoop 默认采用 Simple 的用户认证模式，具有很大的安全风险。

默认的 Simple 认证模式，会在 Hadoop 的客户端执行 whoami 命令，并以 whoami 命令的形式返回结果,作为访问 Hadoop 的用户名(准确地说,是以 whoami 的形式返回结果，作为 Hadoop RPC 的 userGroupInformation 参数发起 RPC Call）。这样会产生以下三个问题。

（1）User Authentication。假设有账号 A 和账号 B，分别在 Host1 和 Host2 上。如果恶意用户在 Host2 上建立了一个同名的账号 A，那么通过 RPC Call 获得的 UGI 就和真正的账号 A 相同，伪造了账号 A 的身份。用这种方式，恶意用户可以访问/修改其他用户的数据。

（2）Service Authentication。Hadoop 采用主从结构，如 NameNode-DataNode、JobTracker-Tasktracker。Slave 节点启动时，主动连接 Master 节点。Slave 到 Master 的连接过程，没有经过认证。假设某个用户在某台非 Hadoop 机器上，错误地启动了一个 Slave 实例，那么也会连接到 Master；Master 会为它分配任务/数据，可能会影响任务的执行。

（3）可管理性。任何可以连到 Master 节点的机器，都可以请求集群的服务，访问 HDFS，运行 Hadoop Job，无法对用户的访问进行控制。

从 Hadoop 0.20.203 开始，社区开发了 Hadoop Security，实现了基于 Kerberos 的 Authentication。任何访问 Hadoop 的用户，都必须持有 KDC（Key Distribution Center）发布的 Ticket 或者 Keytab File（准确地说，是 Ticket Granting Ticket），才能调用 Hadoop 的服务。用户通过密码，获取 Ticket，Hadoop Client 在发起 RPC Call 时读取 Ticket 的内容，使用其中的 Principal 字段，作为 RPC Call 的 UserGroupInformation 参数，解决了问题（1）。Hadoop 的任何 Daemon 进程在启动时，都需要使用 Keytab File 做 Authentication。因为 Keytab File 的分发是由管理员控制的，所以解决了问题（2）。最后，不论是 Ticket，还是 Keytab File，都由 KDC 管理/生成，而 KDC 由管理员控制，解决了问题（3）。

在使用了 Hadoop Security 之后，只有通过了身份认证的用户才能访问 Hadoop，大大增强了数据的安全性和集群的可管理性。之后我们基于 Hadoop Secuirty,与 DW 部门一起开发了 ACL 系统,用户可以自助申请 Hive 上表的权限。在申请通过审批工作流之后，就可以访问了。

JDBC 是一种很常用的数据访问接口，Hive 自带了 Hive Server，可以接受 Hive JDBC Driver 的连接。实际 上，Hive JDBC Driver 是将 JDBC 的请求转化为 Thrift Call 发给 Hive Server,再由 Hive Server 将 Job 启动起来。但 Hive 自带的 Hive Server 并不支持 Security,默认会使用启动 Hive Server 的用户作为 Job 的 owner 提交到 Hadoop，造成安全漏洞。因此，我们自己开发了 Hive Server 的 Security，解决了这个问题。

但在 Hive Server 的使用过程中，我们发现 Hive Server 并不稳定，而且存在内存泄漏。更严重的是由于 Hive Server 自身的设计缺陷，不能很好地应对并发访问的情况，所以我们现在并不推荐使用 Hive JDBC 的访问方式。

社区后来重新开发了 Hive Server 2，解决了并发的问题，我们正在对 Hive Server 2 进行测试。

有一些同事，特别是 BI 的同事，不熟悉以 CLI 的方式使用 Hive，希望 Hive 可以有个 GUI 界面。在上线 Hive Server 之后，我们调研了开源的 SQL GUI Client——Squirrel，可惜使用 Squirrel 访问 Hive 存在一些问题。

- 办公网与线上环境是隔离的，在办公机器上运行的 Squirrel 无法连到线上环境的 Hive Server。
- Hive 会返回大量的数据，特别是当用户对于 Hive 返回的数据量没有预估的情况下，Squirrel 会吃掉大量的内存，然后 Out of Memory 挂掉。
- Hive JDBC 实现的 JDBC 不完整，导致 Squirrel 的 GUI 中只有一部分功能可用，用户体验非常差。

基于以上考虑，我们自己开发了 Hive Web，让用户通过浏览器就可以使用 Hive。Hive Web 最初是作为大众点评第一届 Hackathon 的一个项目被开发出来的，技术上很简单，但获得了良好的反响。现在 Hive Web 已经发展成了一个 RESTful 的 Service，称为 Polestar（https://github.com/dianping/polestar）。

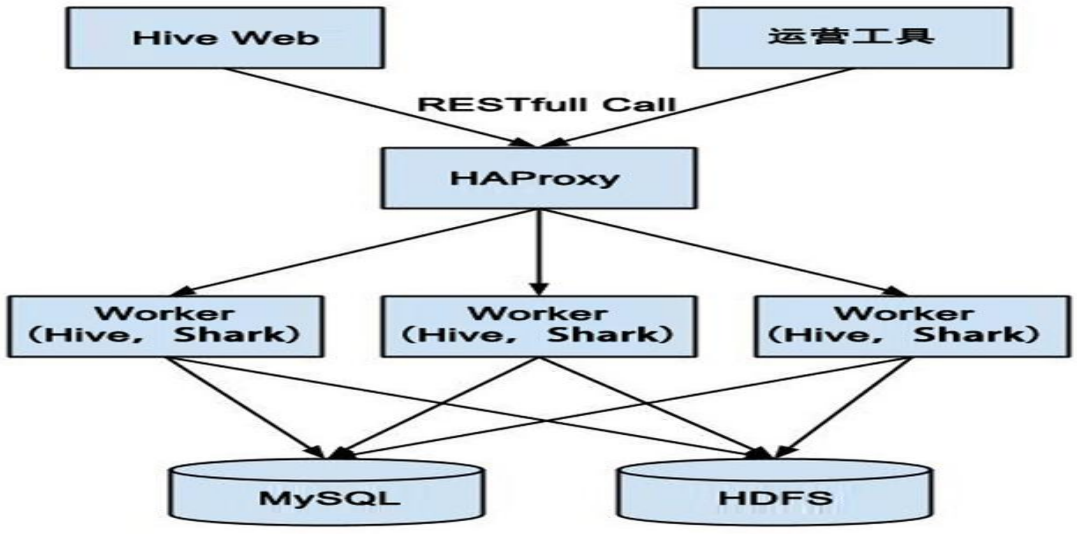


图 3 Polestar 的结构

图3是Polestar的结构图。目前Hive Web只是一个GWT的前端，通过HAProxy将RESTfull Call分发到执行引擎Worker执行。Worker将自身的状态保存在MySQL，将数据保存在HDFS，并使用JSON返回数据或数据在HDFS的 路径。我们还将Shark与Hive Web集成到了一起，用户可以选择以Hive或者Shark执行Query。

一开始我们使用LZO作为存储格式，使大文件可以在MapReduce处理中被切分，提高并行度。但LZO的压缩比不够高，按照我们的测试，Lzo压缩的文件，压缩比基本只有Gz的一半。

经过调研，我们将默认存储格式替换成RCFile，在RCFile内部再使用Gz压缩，这样既可保持文件可切分的特性，同时又可获得Gz的高压缩比，而且因 为RCFile是一种列存储的格式，所以对于不需要的字段就不用从I/O读入，从而提高了性能。图4显示了将Nginx数据分别用Lzo、 RCFile+Gz、RCFfile+Lzo压缩，再不断增加Select的Column数，在Hive上消耗的CPU时间（越小越好）

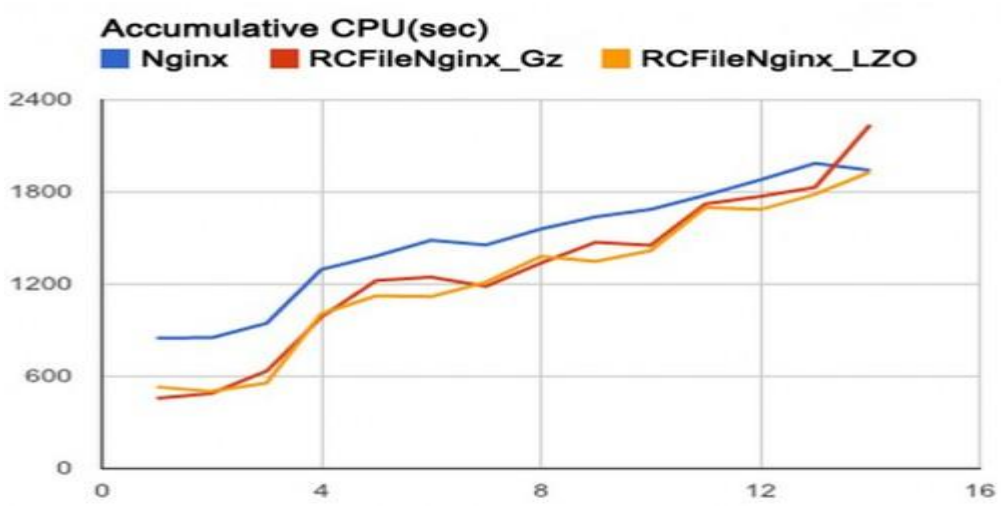


图4 几种压缩方式在Hive上消耗的CPU时间

但RCFile的读写需要知道数据的Schema，而且需要熟悉Hive的Ser/De接口。为了让MapReduce Job能方便地访问RCFile，我们使用了Apache Hcatalog。

社区又针对Hive 0.11开发了ORCFile，我们正在对ORCFile进行测试。

随着Facebook、淘宝等大公司成功地在生产环境应用HBase，HBase越来越受到大家的关注，我们也开始对HBase进行测试。通过测试我们发现 HBase非常依赖参数的调整，在默认配置下，HBase能获得很好的写性能，但读性能不是

特别出色。通过调整 HBase 的参数，在 5 台机器的 HBase 集群上，对于 1KB 大小的数据，也能获得 5 万左右的 TPS。在 HBase 0.94 之后，HBase 已经优化了默认配置。

原来我们希望 HBase 集群与主 Hadoop 集群共享 HDFS，这样可以简化运维成本。但在测试中，发现即使主 Hadoop 集群上没有任何负载，HBase 的性能也很糟糕。我们认为，这是由于大量数据属于远程读写所引起的。所以我们现在的 HBase 集群都是单独部署的。并且通过封装 HBase Client 与 Master-Slave Replication，使用 2 套 HBase 集群实现了 HBase 的 HA，用来支撑线上业务。

2013 年持续演进

在建立了公司主要的大数据架构后，我们上线了 HBase 的应用，并引入 Spark/Shark 以提高 Ad Hoc Query 的执行时间，并调研分布式日志收集系统，来取代手工脚本做日志导入。

现在 HBase 上线的应用主要有 OpenAPI 和手机团购推荐。OpenAPI 类似于 HBase 的典型应用 Click Stream，将开放平台开发者的访问日志记录在 HBase 中，通过 Scan 操作，查询开发者在一段时间内的 Log，但这一功能目前还没有对外开放。手机 团购推荐是一个典型的 KVDB 用法，将用户的历史访问行为记录在 HBase 中，当用户使用手机端访问时，从 HBase 获得用户的历史行为数据，做团购推 荐。

当 Hive 大规模使用之后，特别是原来使用 OLAP 数据库的 BI 部门的同事转入后，一个越来越大的抱怨就是 Hive 的执行速度。对于离 线的 ETL 任务，Hadoop/Hive 是一个良好的选择，但动辄分钟级的响应时间，使得 Ad Hoc Query 的用户难以忍受。为了提高 Ad Hoc Query 的响应时间，我们将目光转向了 Spark/Shark。

Spark 是美国加州大学伯克利分校 AMPLab 开发的分布式计算系统，基于 RDD（Resilient Distributed Dataset），主要使用内存而不是硬盘，可以很好地支持迭代计算。因为是一个基于 Memory 的系统，所以在数据量能够放进 Memory 的情况下，能 够大幅缩短响应时间。Shark 类似于 Hive，将 SQL 解析为 Spark 任务，并且 Shark 复用了大量 Hive 的已有代码。

在 Shark 接入之后，大大降低了 Ad Hoc Query 的执行时间。比如 SQL 语句：

```
select host, count(1) from HIPPOLOG where dt = '2013-08-28' group by host order by host desc;
```

在 Hive 执行的时间是 352 秒，而 Shark 只需要 60~70 秒。但对于 Memory 中放不下的大数据量，Shark 反而会变慢。

目前用户需要在 Hive Web 中选择使用 Hive 还是 Shark，未来我们会在 Hive 中添加 Semantic-AnalysisHook，通过解析用户提交的 Query，根据 数据量的大小，自动选择 Hive 或者 Shark。另外，因为我们目前使用的是 Hadoop 1，不支持 YARN，所以我们单独部署了一个小集群用于 Shark 任务的执行。

Wormhole 解决了结构化数据的交换问题，但对于非结构化数据，例如各种日志，并不适合。我们一直采用脚本或用户程序直接写 HDFS 的方式将用户的 Log 导入 HDFS。缺点是，需要一定的开发和维护成本。我们 希望使用 Apache Flume 解决这个问题，但在测试了 Flume 之后，发现了 Flume 存在一些问题：Flume 不能保证端到端的数据完整性，数据可能丢失，也可能重复。

例如，Flume 的 HDFSsink 在数据写入/读出 Channel 时，都有 Transcation 的保证。当 Transaction 失败时，会回滚，然后重试。但由于 HDFS 不可修改文件的内容，假设有 1 万行数据要写入 HDFS，而在写入 5000 行时，网络出现问题导致写入失败，Transaction 回滚，然后重写这 10000 条记录成功，就会导致第一次写入的 5000 行重复。我们试图修正 Flume 的这些问题，但由于这些问题是设计上的，并不能通过简单的 Bugfix 来解决，所以我们转而开发 Blackhole 系统将数据流导入 HDFS。目前 Blackhole 正在开发中。

总结

图 5 是各系统总体结构图，深蓝部分为自行开发的系统。

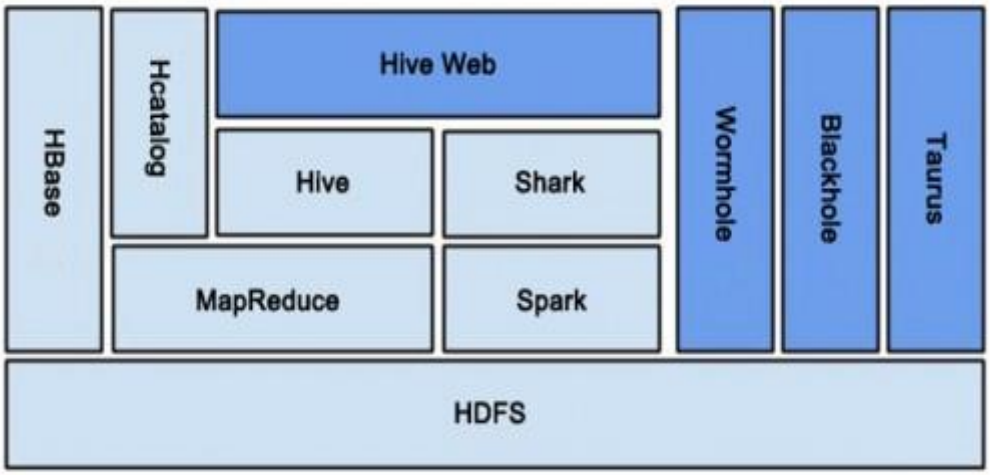


图 5 大众点评各系统总体结构图

在这 2 年多的 Hadoop 实践中，我们得到了一些宝贵经验。

- 建设一支强大的技术团队是至关重要的。Hadoop 的生态系统，还处在快速演化中，而且文档相当匮乏。只有具备足够强的技术实力，才能用好开源软件，并在开源软件不能满足需求时，自行开发解决问题。
- 要立足于解决用户的需求。用户需要的东西，会很容易被用户接受，并推广开来；某些东西技术上很简单，但可以解决用户的大问题。
- 对用户的培训，非常重要。

原文：
http://www.csdn.net/article/2013-12-18/2817838-big-data-practice-in-dianping?utm_source=tuicool

MetaQ 在双十二彩票中的运用

双十二大促是淘宝集市的年终促销活动，活动当天扫描首页二维码赠送一注彩票的活动更是让大家“玩”了一把。面对瞬间的数倍于往常的峰值，如何让用户有一个良好的体验，如何保证系统的稳定运行，让我们来揭秘这一切。

归纳一下系统需要做到如下几点：

- RT 足够短
- 压力分布均匀
- 复杂逻辑分离，异步化

系统结构图

图 1-系统结构图

大体分为两个部分：活动系统，彩票系统，他们之间通过消息驱动。
活动系统里面只更新一个彩票分配的状态，数据更新成功就返回给用户，逻辑足够简单能保证 RT 非常短。彩票系统负责真正的彩票出票和更新用户状态等一些耗时操作。
用户在首页扫描二维码发起对活动系统的请求，活动系统更新彩票分配状态，产生一条消息，立即返回并。彩票系统收到这个消息完成后续出票等一些的业务操作。整个过程是通过 MetaQ 提供的消息服务驱动完成。活动的时候会有大量的请

求涌进活动系统，会产生大量的消息，预估 qps 达到 24w；并且消息不可丢失，确保用户都能领到一注彩票；活动系统，彩票系统业务复杂度相差较大，处理能力也相差较大，可能会出现大量的堆积；为了让用户有个较好的体验，消息的消费也需要足够的及时。综上对 MetaQ 有这么一些挑战：

- 高吞吐量
- 数据可靠
- 高效堆积
- 消息投递足够低延迟

下面介绍一下 MetaQ 如何做到这些的。

MetaQ 简介

METAQ 是一款完全的队列模型消息中间件，服务器使用 Java 语言编写，可在多种软硬件平台上部署。客户端支持 Java、C++编程语言，已于 2012 年 3 月对外开源，开源[地址](http://metaq.taobao.org/)。MetaQ 的设计目标是高吞吐量，高效堆积。完全的队列模型还提供了顺序消息，消息回溯等一些特性。

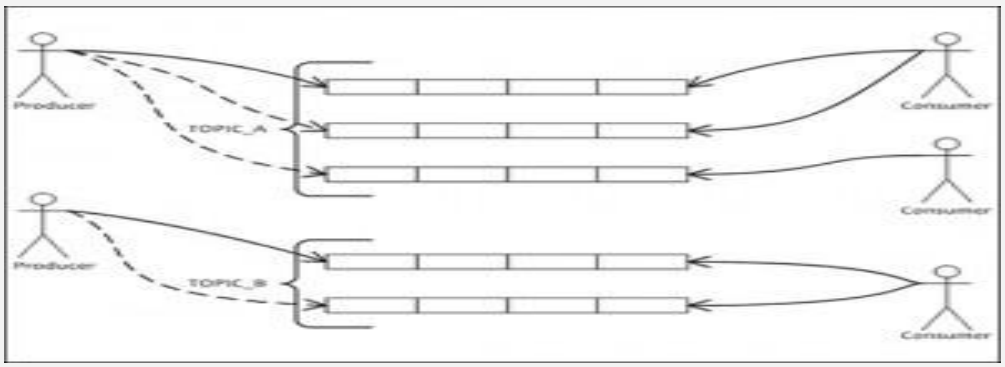


图 2-MetaQ 整体结构

基本概念

- Topic 消息主题
- Partition 分区,代表一个消费队列（一个 Topic 可以划分为多个分区，分区数越多并行度越大，支持的 Qps 越高）
- Group 消费分组，代表一个消费集群

MetaQ 对外提供的是一个队列服务，内部实现也是完全的队列模型，这里的队列是持久化的磁盘队列，具有非常高的可靠性，并且充分利用了操作系统 cache 来提高性能。这些特性都源于存储层的设计。

MetaQ 存储结构

MetaQ 的存储结构是根据阿里大规模互联网应用需求，完全重新设计的一套存储结构,使用这套存储结构可以支持上万的队列模型,并且可以支持消息查询、分布式事务、定时队列等功能，如图 3 所示。

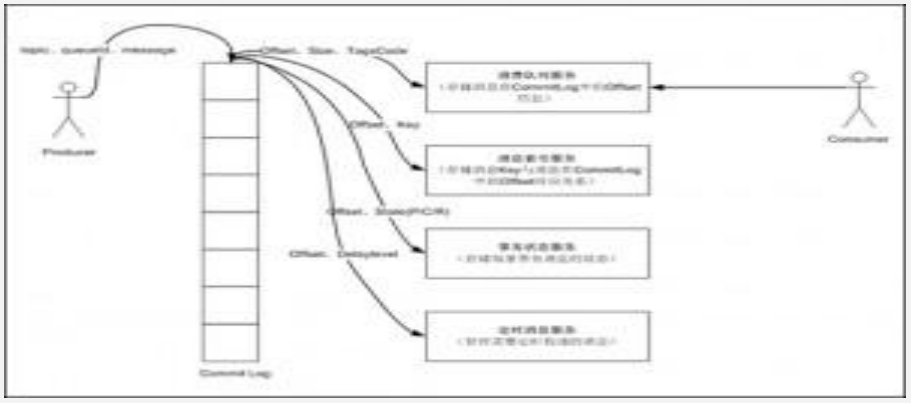


图 3-MetaQ 存储体系

存储层可以大致分为数据文件（CommitLog）和索引文件两部分。数据文件保存了所有的消息的内容，索引文件保存了消息所在数据文件的偏移量。消息数据不区分 Topic，顺序的 append 到 CommitLog，索引文件按照 Topic-Partition 维度组织，不同分区的消息 append 到不同索引队列里面。

消息写入

客户端发送一条消息,数据首先会写到文件缓存中，同时派发一个写索引请求；异步的构建消息索引。

消息读取

客户端读取消息，根据索引的位置找到需要读取的消息的物理位置和消息长度，从 CommitLog 中读取数据，通过文件缓存来加速消息的读取。通常热数据都在缓存中，无需 IO 操作；非热数据，会触发缺页中断，数据从磁盘加载到文件缓存中，直接写到 socket 缓冲区，避免数据进入 Java 堆。

数据刷盘

刷盘后数据最终持久化到磁盘。刷盘方式分为两种方式：同步刷盘，数据写入缓存后立即刷盘，确保数据落盘后返回客户端，MetaQ 在同步刷盘过程中也做了一定优化避免过多的性能损失；异步刷盘，数据批量定时的刷到磁盘。

数据清理

消息数据按照文件有效期定时做清理。

数据复制

数据可靠性要求很高的应用，可通过数据复制保证数据的可靠。MetaQ 提供两种数据同步的方式：同步双写，数写入到主机后会同时写到备机，主备都写成功才返回客户端成功，主备间数据无延迟，MetaQ 有一套自己的主备间高效数据复制方案；异步复制，数据写到主机后返回客户端成功，主备间异步同步，主备间存在一定的延迟。

MetaQ 单机上万队列

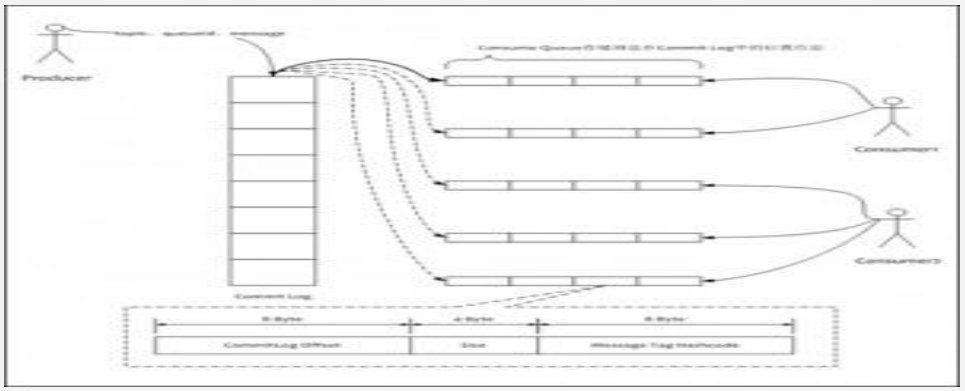


图 4-MetaQ 单机上万队列

MetaQ 的大部分功能都是靠队列来驱动，以文件的方式存储，通过内存映射对数据进行操作。所有的消息都是顺序的写到数据文件（CommitLog），完全顺序的写入，避免随机 IO；消息索引按照 Topic 和 Partition 的维度区分串行的写到索引文件。通过这种这种组织方式可以实现单机上万队列，如图 4 所示。

数据流图

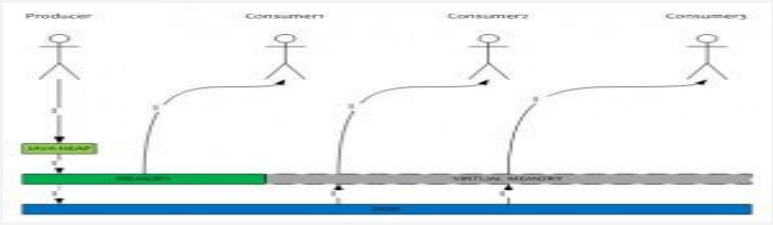


图 5-MetaQ 数据流图

消息数据首先写入到 Java 堆，然后在写入到文件缓存，在根据具体的刷盘策略 Flush 到磁盘;消费消息的时候如果是热数据则直接从系统缓存写到 socket 发到远端;如果非热数据则由系统产生缺页中断将数据从磁盘加载到系统缓存在写到 socket，数据不进应用程序内存空间。内存的管理，页面的换入换出都是由 OS 来管理的。同时消息的拉取也是批量的，一次处理多条数据，尽量减少往返的时间。

MetaQ 性能依赖于系统内存分配，磁盘 IO 的有效利用，所以我们也对操作系统内存分配，脏页会写策略，以及 IO 调度算法做了一些调优，让资源的分配耗时趋于平稳，堆积的时候保持较高的吞吐量。

负载均衡

发送端负载

默认发送端通过轮询的方式向 broker 写消息，如图 6 所示；也可以自行指定消息发到哪里。

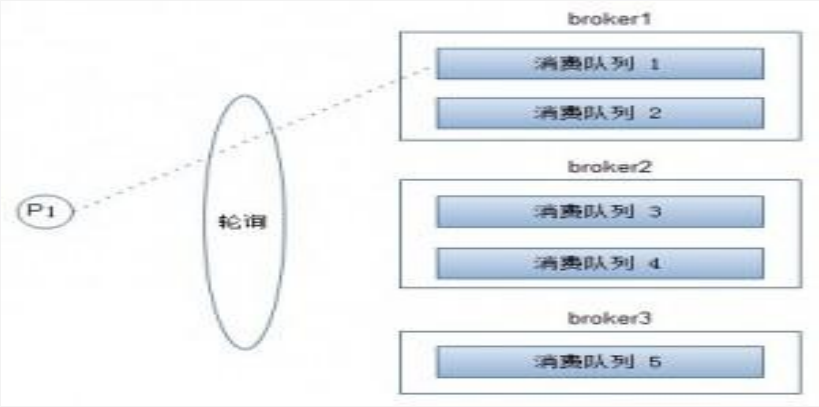


图 6-默认发送端负载

消费端负载

默认是消费集群机器均分所有的消费队列。余下的部分由靠前的消费者消费，如图 7 所示。消费端负载均很也可以定制，如同机房优先等。

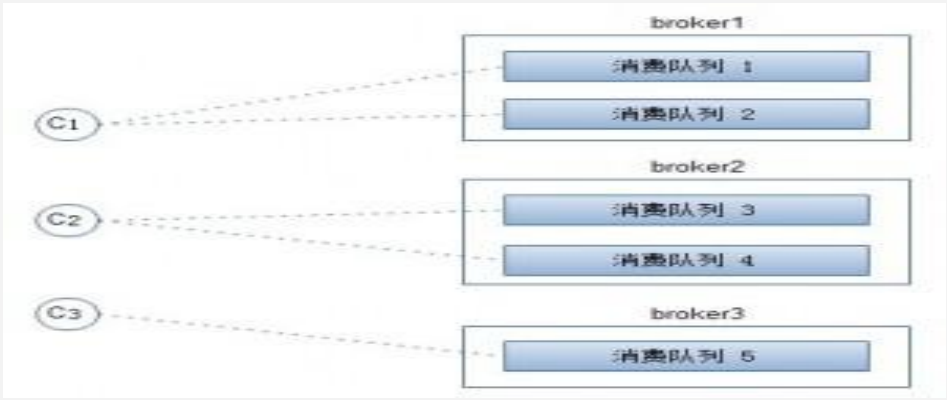


图 7-默认消费端负载

其他特性

综上 MetaQ 主要有以下一些特性：

- 高吞吐量，低延迟
- 高效堆积；亿级消息堆积能力
- 完全分布式；客户端，服务端都是分布式的，每个结点的变化都能动态负载
- 顺序消息；能保证消息的严格顺序
- 实时消息订阅
- 丰富的消息拉取模式
- 消息过滤
- 消息内容查询
- 消息回溯
- 事物消息
- 高可用 HA

上文介绍了双十二彩票系统面临的挑战，以及 MetaQ 如何克服这些难点。如果对其中的实现细节感兴趣；或者对其他的一些特性感兴趣可以参看 MetaQ 开源地址：[http://metaq.taobao.org/]。有什么疑问或者好的建议也欢迎大家给予我们反馈。

原文：http://jm-blog.aliapp.com/?p=3405&utm_source=tuicool

Redis 在新浪微博中的应用

Redis 简介

1. 支持 5 种数据结构

支持 strings, hashes, lists, sets, sorted sets
string 是很好的存储方式，用来做计数存储。sets 用于建立索引库非常棒；

2. K-V 存储 vs K-V 缓存

新浪微博目前使用的 98%都是持久化的应用，2%的是缓存，用到了 600+服务器
Redis 中持久化的应用和非持久化的方式不会差别很大：
非持久化的为 8-9 万 tps，那么持久化在 7-8 万 tps 左右；
当使用持久化时，需要考虑到持久化和写性能的配比，也就是要考虑 redis 使用的内存大小和硬盘写的速率的比例计算；

3. 社区活跃

Redis 目前有 3 万多行代码，代码写的精简，有很多巧妙的实现，作者有技术洁癖
Redis 的社区活跃度很高，这是衡量开源软件质量的重要指标，开源软件的初期一般都没有商业技术服务支持，如果没有活跃社区做支撑，一旦发生问题都无处求救；

Redis 基本原理

redis 持久化(aof) append online file：
写 log(aof)，到一定程度再和内存合并. 追加再追加，顺序写磁盘，对性能影响非常小

1. 单实例单进程

Redis 使用的是单进程，所以在配置时，一个实例只会用到一个 CPU；
在配置时，如果需要让 CPU 使用率最大化，可以配置 Redis 实例数对应 CPU 数，Redis 实例数对应端口数(8 核 Cpu，8 个实例，8 个端口)，以提高并发:
单机测试时，单条数据在 200 字节，测试的结果为 8~9 万 tps；

2. Replication

过程：数据写到 master-->master 存储到 slave 的 rdb 中-->slave 加载 rdb 到内存。
存储点(save point)：当网络中断了，连上之后，继续传.
Master-slave 下第一次同步是全传，后面是增量同步；、

3. 数据一致性

长期运行后多个结点之间存在不一致的可能性；
开发两个工具程序：
1.对于数据量大的数据，会周期性的全量检查；
2.实时的检查增量数据，是否具有一致性；

对于主库未及时同步从库导致的不一致，称之为延时问题；
对于一致性要求不是那么严格的场景，我们只需要要保证最终一致性即可；
对于延时问题，需要根据业务场景特点分析，从应用层面增加策略来解决这个问题；例如：
1.新注册的用户，必须先查询主库；
2.注册成功之后，需要等待 3s 之后跳转，后台此时就是在做数据同步。

新浪 Redis 使用历程

2009 年，使用 memcache(用于非持久化内容)，memcacheDB(用于持久化+计数)，memcacheDB 是新浪在 memcache 的基础上，使用 BerkeleyDB 作为数据持久化的存储实现；

1. 面临的问题

- 数据结构(Data Structure)需求越来越多，但 memcache 中没有，影响开发效率

- 性能需求，随着读操作的量的上升需要解决，经历的过程有:
  数据库读写分离(M/S)-->数据库使用多个 Slave-->增加 Cache (memcache)-->转到 Redis
- 解决写的问题：
  水平拆分，对表的拆分，将有的用户放在这个表，有的用户放在另外一个表；
- 可靠性需求
  Cache 的"雪崩"问题让人纠结
  Cache 面临着快速恢复的挑战
- 开发成本需求
  Cache 和 DB 的一致性维护成本越来越高(先清理 DB，再清理缓存，不行啊，太慢了!)
  开发需要跟上不断涌入的产品需求
  硬件成本最贵的就是数据库层面的机器，基本上比前端的机器要贵几倍，主要是 IO 密集型，很耗硬件；
- 维护性复杂
  一致性维护成本越来越高；
  BerkeleyDB 使用 B 树，会一直写新的，内部不会有文件重新组织；这样会导致文件越来越大;大的时候需要进行文件归档,归档的操作要定期做；这样，就需要有一定的 down time；

基于以上考虑， 选择了 Redis

2. 寻找开源软件的方式及评判标准

- 对于开源软件，首先看其能做什么，但更多的需要关注它不能做什么，它会有什么问题？
- 上升到一定规模后，可能会出现什么问题，是否能接受？
- google code 上，国外论坛找材料(国内比国外技术水平滞后 5 年)
- 观察作者个人的代码水平

Redis 应用场景

1. 业务使用方式

- hash sets: 关注列表，粉丝列表，双向关注列表(key-value(field)，排序)

- string(counter)：微博数，粉丝数，...(避免了 select count(*) from ...)
- sort sets(自动排序)：TopN，热门微博等，自动排序
- lists(queue)：push/sub 提醒,...

上述四种，从精细化控制方面，hash sets 和 string(counter)推荐使用，sort sets 和 lists(queue)不推荐使用
还可通过二次开发，进行精简。比如：存储字符改为存储整形，16 亿数据，只需要 16G 内存
存储类型保存在 3 种以内，建议不要超过 3 种；

将 memcache +myaql 替换为 Redis：
Redis 作为存储并提供查询，后台不再使用 mysql，解决数据多份之间的一致性问题；

2. 对大数据表的存储

（eg：140 字微博的存储）
一个库就存唯一性 id 和 140 个字；
另一个库存 id 和用户名，发布日期、点击数等信息，用来计算、排序等，等计算出最后需要展示的数据时再到第一个库中提取微博内容；

改进的 3 个步骤:
1）发现现有系统存在问题;
2）发现了新东西，怎么看怎么好，全面转向新东西;
3）理性回归，判断哪些适合新东西，哪些不适合，不合适的回迁到老系统

3. 一些技巧

- 很多应用，可以承受数据库连接失败，但不能承受处理慢
- 一份数据，多份索引(针对不同的查询场景)
- 解决 IO 瓶颈的唯一途径：用内存
- 在数据量变化不大的情况下，优先选用 Redis

遇到的问题及解决办法

(注意：都是量特别大时候会出现的，量小了怎么都好说)

1.Problem: Replication 中断后，重发—>网络突发流量

Solution: 重写 Replication 代码，rdb+aof(滚动)

2.Problem: 容量问题

Solution: 容量规划和 M/S 的 sharding 功能(share nothing，抽象出来的数据对象之间的关联数据很小)
增加一些配置，分流，比如：1,2,3,4，机器 1 处理%2=1 的，机器 2 处理%2=0 的.

低于内存的 1/2 使用量，否则就扩容（建议 Redis 实例使用的数据，最大不要超过内存的 80%）

我们线上 96G/128G 内存服务器不建议单实例容量大于 20/30G。

微博应用中单表数据最高的有 2T 的数据,不过应用起来已经有些力不从心；每个的端口不要超过 20G；测试磁盘做 save 所需要的时间，需要多长时间能够全部写入；内存越大，写的时间也就越长；

单实例内存容量较大后,直接带来的问题就是故障恢复或者 Rebuild 从库的时候时间较长，对于普通硬盘的加载速度而言，我们的经验一般是 redis 加载 1G 需要 1 分钟；（加载的速度依赖于数据量的大小和数据的复杂度）
Redis rewrite aof 和 save rdb 时，将会带来非常大且长的系统压力，并占用额外内存，很可能导致系统内存不足等严重影响性能的线上故障。reblance：现有数据按照上述配置重新分发。
后面使用中间层，路由 HA；
注：目前官方也正在做这个事，Redis Cluster，解决 HA 问题；

3. Problem: bgsave or bgwriteaof 的冰晶问题

Solution: 磁盘性能规划和限制写入的速度，比如：规定磁盘以 200M/s 的速度写入，细水长流，即使到来大量数据. 但是要注意写入速度要满足两个客观限制:
符合磁盘速度
符合时间限制(保证在高峰到来之前，就得写完)

4.Problem: 运维问题

1）Inner Crontab：把 Crontab 迁移到 Redis 内部，减少迁移时候的压力
本机多端口避免同时做 - 能做到

同一业务多端口(分布在多机上)，避免同时做 - 做不到

2）动态升级：先加载.so 文件，再管理配置，切换到新代码上(Config set 命令)

把对 redis 改进的东西都打包成 lib.so 文件，这样能够支持动态升级

自己改的时候要考虑社区的升级。当社区有新的版本，有很好用的新功能时，要能很容易的与我们改进后的版本很好的 merge；升级的前提条件：模块化，以模块为单位升级加载时间取决于两个方面：数据大小，数据结构复杂度. 一般，40G 数据耗时 40 分钟分布式系统的两个核心问题：A.路由问题 B.HA 问题

3）危险命令的处理：比如：fresh all 删除全部数据，得进行控制

运维不能只讲数据备份，还得考虑数据恢复所需要的时间；

增加权限认证(管理员才有权限)eg：flashall 权限认证，得有密码才能做；

当然，高速数据交互一般都不会在每次都进行权限认证，通用的处理策略是第一次认证，后期都不用再认证；控制 hash 策略(没有 key，就找不到 value；不知道 hash 策略，就无法得到 key)

4）Config Dump:内存中的配置项动态修改过，按照一定策略写入到磁盘中(Redis 已支持)

5)bgsave 带来 aof 写入很慢:fdatasync 在做 bgsave 时，不做 sync aof(会有数据出入)

6）成本问题：(22T 内存，有 10T 用来计数)Redisscounter(16 亿数据占用 16G 内存) - 全部变为整型存储，其余(字符串等)全不要

Redis+SSD(counterService 计数服务)顺序自增，table 按照顺序写，写满 10 个 table 就自动落地(到 SSD)

存储分级：内存分配问题，10K 和 100K 写到一块，会有碎片. Sina 已经优化到浪费只占 5%以内(已经很好了!)

5.Problem: 分布式问题

1.Config Server：命名空间，特别大的告诉访问，都不适合用代理，因为代理降低速度，但是，Sina 用了(单机多端口，Redis Cluster，sentinel)

Config Server 放到 Zookeeper 上最前面是命名服务，后面跟的是无状态的 twmemproxy（twitter 的改进的，用 C 写的），后面才是 redis；

2.twmemproxy

应用不必关心连接失败，由代理负责重连

把 Hash 算法放到代理商

代理后边的升级，前端不关心，解决了 HA 的问题
无状态，多台代理无所谓

3. AS --> Proxy -->Redis

4. Sina 的 Redis 都是单机版，而 Redis-Cluster 交互过于复杂，没有使用做 HA 的话，一定要配合监控来做，如果挂了之后，后续该如何做；
并不是追求单机性能，而是集群的吞吐量，从而可以支持无线扩展；

经验总结

- 提前做好数据量的规划，减少 sharding(互联网公司一般以年为单位)
- 只存精细化数据(内存很金贵!)
- 存储用户维度的数据
  对象维度的数据要有生命周期
  特别是数据量特别大的时候，就很有必要来进行划分了；
- 暴露服务的常见过程：IP-->负载均衡-->域名-->命名服务(一张表：名字+资源(IP+端口))
- 对于硬件消耗，IO、网络和 CPU 相比，Redis 最消耗的是 CPU，复杂的数据类型必定带来 CPU 消耗；
- 新浪微博响应时间超时目前设置为 5s；（返回很慢的记录 key，需记录下来分析，慢日志）；
- 备份的数据要定期要跑一下生产的数据；用来检查备份数据的有效性；
- slave 挂多了肯定会对 master 造成比较的影响;新浪微博目前使用的 M/S 是一拖一，主要用来做容灾；
  同步时，是 fork 出一个单独进程来和 slave 进行同步；不会占用查询的进程；
- 升级到 2.6.30 以后的 linux 内核；
  在 2.6.30 以上对软中断的问题处理的很好，性能提升效果明显，差不多有 15%到 30%的差距；
- redis 不用读写分离，每个请求都是单线程，为什么要进行读写分离。

原文：
http://blog.csdn.net/liudong1105/article/details/17440711?utm_source=tuicool

RocksDB 介绍 :一个比 LevelDB 更彪悍的引擎

关于 LevelDB 的资料网上还是比较丰富的，如果你尚未听说过 LevelDB，那请稍微预习一下，因为 RocksDB 实际上是在 LevelDB 之上做的改进。本文主要侧重在架构上对RocksDB对LevelDB改进的地方做个简单介绍并添加一些个人的看法，更详细的信息读者可参考其官网：http://rocksdb.org/

RocksDB 是在 LevelDB 原来的代码上进行改进完善的，所以在用法上与 LevelDB 非常的相似。如下，就是简单的把原来 Leveldb 信息替换为 Rocksdb，从继承的角度看，Rocksdb 就像是 Leveldb 的后辈。

RocksDB:

```
#include "rocksdb/db.h"

rocksdb::DB* db;
rocksdb::Options options;
options.create_if_missing = true;

rocksdb::Status status = rocksdb::DB::Open(options, "/tmp/testdb", &db);

assert(status.ok());

status = db->Get(rocksdb::ReadOptions(), key1, &value);
status = db->Put(rocksdb::WriteOptions(), key2, value);
status = db->Delete(rocksdb::WriteOptions(), key1);

delete db;

```

LevelDB:

```
#include "leveldb/db.h"

leveldb::DB *db;
leveldb::Options options;
options.create_if_missing = true;

leveldb::Status status = leveldb::DB::Open(options, "/tmp/testdb", &db);

assert(status.ok());

status = db->Get(leveldb::ReadOptions(), key1, &value);
status = db->Put(leveldb::WriteOptions(), key2, value);
status = db->Delete(leveldb::WriteOptions(), key1);

delete db;

```

RocksDB 虽然在代码层面上是在 LevelDB 原有的代码上进行开发的，但却借鉴了 Apache HBase 的一些好的 idea，要知道 LevelDB 启发也是来自于 HBase 的，所以有 LevelDB 是 HBase RegionServer 的简单实现之说。在云计算横行的年代，开口不离 Hadoop，RocksDB 也开始支持 HDFS，允许从 HDFS 读取数据。而 LevelDB 则是一个比较单一的存储引擎，有点我就是我，除了我依然只有我的感觉。也是因为 LevelDB 的单一性，在做具体的应用的时候一般需要对其作进一步扩展。

RocksDB 支持一次获取多个 K-V，还支持 Key 范围查找。LevelDB 只能获取单个 Key

RocksDB 除了简单的 Put、Delete 操作，还提供了一个 Merge 操作，说是为了对多个 Put 操作进行合并。站在引擎实现者的角度来看，相比其带来的价值，其实现的成本要昂贵很多。个人觉得有时过于追求完美不见得是好事，据笔者所测（包括测试自己编写的引擎），性能的瓶颈其实主要在合并上，多一次少一次 Put 对性能的影响并无大碍。

RocksDB 提供一些方便的工具，这些工具包含解析 sst 文件中的 K-V 记录、解析 MANIFEST 文件的内容等。有了这些工具，就不用再像使用 LevelDB 那样，只能在程序中才能知道 sst 文件 K-V 的具体信息了。

RocksDB 支持多线程合并，而 LevelDB 是单线程合并的。LSM 型的数据结构，最大的性能问题就出现在其合并的时间损耗上，在多 CPU 的环境下，多线程合并那是 LevelDB 所无法比拟的。不过据其官网上的介绍，似乎多线程合并还只是针对那些与下一层没有 Key 重叠的文件，只是简单的 rename 而已，至于在真正数据上的合并方面是否也有用到多线程，就只能看代码了。

RocksDB 增加了合并时过滤器，对一些不再符合条件的 K-V 进行丢弃，如根据 K-V 的有效期进行过滤。

压缩方面 RocksDB 可采用多种压缩算法，除了 LevelDB 用的 snappy，还有 zlib、bzip2。LevelDB 里面按数据的压缩率（压缩后低于 75%）判断是否对数据进行压缩存储，而 RocksDB 典型的做法是 Level 0-2 不压缩，最后一层使用 zlib，而其它各层采用 snappy。

在故障方面，RocksDB 支持增量备份和全量备份，允许将已删除的数据备份到指定的目录，供后续恢复。

RocksDB 支持在单个进程中启用多个实例，而 LevelDB 只允许单个实例。

RocksDB 支持管道式的 Memtable，也就说允许根据需要开辟多个 Memtable，以解决 Put 与 Compact 速度差异的性能瓶颈问题。在 LevelDB 里面因为只有一个 Memtable，如果 Memtable 满了却还来不及持久化，这个时候 LevelDB 将会减缓

Put 操作，导致整体性能下降。笔者目前写的引擎在这方面竟然跟 RocksDB 不谋而合，这里偷偷乐一下，呵呵。

看完上面这些介绍，相比 LevelDB 是不是觉得 RocksDB 彪悍的不可思议，很多该有的地方都有，该想的都想到了，简直不像在做引擎库，更像是在做产品。不过虽然 RocksDB 在性能上提升了不少，但在文件存储格式上跟 LevelDB 还是没什么变化的， 稍微有点更新的只是 RocksDB 对原来 LevelDB 中 sst 文件预留下来的 MetaBlock 进行了具体利用。

个人觉得 RocksDB 尚未解决的地方：

1. 依然是完全依赖于 MANIFEST，一当该文件丢失，则整个数据库基本废掉。
2. 合并上依然是整个文件载入，一些没用的 Value 将被多次的读入内存，如果这些 Value 很大的话，那没必要的内存占用将是一个可观的成本。

关于这两个问题，尤其是后面那个问题，笔者已有相应的解决方案，至于结果如何只等日后实现之后再作解说了。

原文：http://tech.uc.cn/?p=2592&utm_source=tuicool

十大怪异的编程语言

人们都认为编程语言很容易使用和学习。编程语言应该给你提供数据结构让你来解决实际问题，它们的语法也应清晰明了，容易理解且执行速度快，没有任何 bug。但有时候编程语言设计者们会创建一些违背这些原则的语言，要么供研究使用要么纯属娱乐。下面是 10 种最怪异的最不切实际的编程语言。

1）LOLCODE

很少有编程语言像 LOLCODE 这样注入了这么多网络流行预言。它由英国兰卡斯特大学研究员 Adam Lindsay 于 2007 年创建，该语言的关键字都是大写的。你几乎可以想象一个猫在生产环境中使用它。

```
HAI
CAN HAS STDIO?
I HAS A VAR
IM IN YR LOOP
   UP VAR!!1
```

```
    VISIBLE VAR
    IZ VAR BIGGER THAN 10? KTHX
  IM OUTTA YR LOOP
  KTHXBYE
```

与该份列表中的大部分语言一样，LOLCODE 没有标准库可言。这意味着你只能使用 LOLCODE 来读取文件或写入文本到控制台而不能干其他的事。如果你需要功能更强的版本，那么看看因 LOLCODE 受到启发的 LOLPython，它可以让你访问所有 Python 的强大的功能库。

更多的代码和例子请看维基百科介绍。

2）Glass

Glass 是一门深奥的编程语言，由 Gregor Richards 于 2005 年发展起来。它将非直观的后缀符号与沉重的面向对象结合起来，需要一个主栈与其面向对象的结构相结合才能进一步玩出花样。据作者所言没有其它的语言像这样执行，因为看起来非常的白痴。

下面是该语言的一个例子，程序输出 Fibonacci 序列：

```
  {F[f(_a)A!(_o)O!(_t)$(_n)1=,(_isle)(_n)*(_a)(le).?=/(_isle)^\(_n)
*(_a)
  s.?(_t)f.?(_n)*(_a)s.?(_t)f.?(_a)a.?]}{M[m(_a)A!(_f)F!(_o)O!(_n)=
(_nlm)
  =/(_nlm)(_n)*(_f)f.?(_o)(on).?"
"(_o)o.?(_n)(_n)*(_a)a.?=(_nlm)(_n)*
  (_a)(le).?=\]}
```

更多的例子和说明请看这个网页。

3) Brainfuck

Brainfuck 是晦涩难懂的语言巨星，受到了大批崇拜者的追捧。该语言是出了名的难以编程，仅有 8 个简单的命令和一个指令指针。它的设计就是为了挑战和娱乐程序员，而不是为了用于实际用途。它由 Urban Müller 于 1993 年创建。

下面是“Hello world!”的例子：

```
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++. >+. +++++++. . +++. >++
.
<<++++++++++++++. >. +++. ------. --------. >+. >.
```

更多请看 这里

4) Chicken

Chicken 不仅是语言的名字，也是该语言允许的唯一关键字。关键字重复的次数和换行决定执行的具体操作。下面是一个例子，你能猜出它是干嘛的吗？

```
chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken
chicken chicken chicken chicken chicken chicken
```

更多请看 这篇文章

5) Whitespace

Whitespace 是一门很出色的编程语言。它仅通过空格、制表符和回车来理解并忽略所有其他字符。这个特性还允许 Whitespace 程序隐藏在其他语言程序的源代码中，例如 JavaScript，Javascript 的缩进就是用空格写的。下面是一个例子用以打印“Hello,
world!”（S 表示单个空格，T 表示制表符）：

```
S S S T S S T   S S S
T
S S S S S T T   S S T   S T
T
S S S S S T T   S T T   S S
T
S S S S S T T   S T T   S S
T
S S S S S T T   S T T   T   T
T
S S S S S T S T T   S S
T
S S S S S T S S S S S
T
S S S S S T T   T   S T T   T
T
S S S S S T T   S T T   T   T
T
S S S S S T T   T   S S T   S
T
S S S S S T T   S T T   S S
T
```

```
S S S S S T T   S S T   S S
T
S S S S S T S S S S T
T
S S
```

学习更多请看 这里

6) ///

///是一门极简派风格的编程语言，仅包含一个操作——即用/source/replacement/这样的形式进行字符串替换。它由 Tanner Swett 于 2008 年发明。该语言的功能十分有限，但是一些聪明的程序员能够将字符串替换转换为完整的工作程序，用以循环和输出数据，下面是一个简单的“Hello, world!”的程序：

```
/ world! world!/Hello,/ world! world! world!
```

学习更多请看 这里

7) Befunge

Befunge 是一个二维编程语言。你的代码放在一个固定大小的场地，该场地的每一列都能存放代码或者数据，你的程序可以替换任何想要替换的列。解释器从左上列开始从左到右解析。你可以用特殊的方向指令控制解释器的方向。例如，下面是一个无限循环的例子：

```
>v
```

```
^<
```

而下面这个是“Hello, world!”的程序：

```
0"!dlroW ,olleH">:#,_@
```

学习更多请看 这里

8）Piet

Piet 编程语言的程序看起来都像抽象画的位图，基本的构建块就是颜色块。它支持 20 种不同的颜色，有些实现支持的更多。编译器由图像周围的移动“指针”引导，在图像中从一个色块移动到下一个。下面是一个“Hello world!”程序：

Hello World in Piet

学习更多请看该语言的 主页

9) Malbolge

Malbolge 语言由 Ben Olmstead 于 1998 年发明，以“但丁的地狱”中地狱的第 8 圈命名。该名字不是随机选取的——该语言设计的初衷就是为了不可能写出有用的编程语言。在创建几年后，有人发现了设计中的漏洞使其能写 Malbolge 程序。你必须得成为一个密码科学家才能用它写出有意义的程序。下面是 Malbolge 中看起来像“Hello World!”程序：

```
    ('&%:9]!~}|z2Vxwv-,POqponl$Hjig%eB@@>}=<M:9wv6WsU2T|nm-,jcL(I&%$#
"
    `CB]V?Tx<uVtT`Rpo3NIF.Jh++FdbCBA@?]!~|4XzyTT43Qsqq(Lnmkj"Fhg${z@>
```

学习更多请看 这里

10) ~English

~English 是一门试图模仿自然声音语言的编程语言，其语法非常宽松，使程序员有很大的自由表达空间。但程序员不能自己定义函数而只能使用该语言内置的函数。下面是一个例子程序：

```
    Display "Hello world!" and a newline.
    Stop the program.
```

学习更多请看 这里，你可能也对 Shakespeare 感兴趣。

荣誉提名：JavaScript

Javascript 是如今最受欢迎的编程语言，但受欢迎带来的副作用就是该语言自身的每处特性，无论该语言多美妙，还是每天给成千上万的程序员带来了 bug，这使得 JS 也列在这里。这里有一个完整的网站列出了 Javascript 的怪异之处，给你一个例子，下面是两个简短的完全有效的 JS 片段（你可以在你的控制台上试验一下）：

```
    // 这将输出 10:
    alert(++[[]][+[]]+[+[]]);

    // 这个则输出 "fail":
    alert((![]+[])[+[]]+(![]+[])[+!+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(
![]+[])[!+[]+!+[]]);
```

更多的信息请看这些 StackOverflow 上面的问题：

http://stackoverflow.com/questions/7202157/can-you-explain-why-10

http://stackoverflow.com/questions/4170978/explain-why-this-works?lq=1

结论

包含在这份列表中的这些深奥的语言，给大家提供了一种提出和验证新思想的方法。有时候这些想法会影响整个工业界。但是它们中的大部分都是不切实际的且深奥、范围狭窄，你刚刚只学了其中的 10 种，但实际上有几百种。

原文：http://segmentfault.com/a/1190000000364832?utm_source=tuicool

2013 年 —— Facebook 在开源方面的工作介绍

自从 Facebook 的第一行 PHP 代码，第一句 MySQL 的 INSERT 语句，开源就已经是我们工程哲学中的一个重要的部分。

现在，我们使用、维护并为大量的主要项目做出了贡献——涉及多种领域如手机工具、大数据系统、客户端的 web 库、后端的运行库和基础架构，也涉及到开放计算项目，服务器和存储硬件。

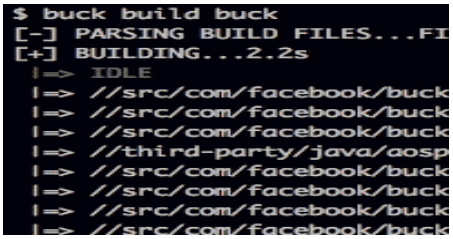
2013 是我们开源项目具有重要意义的一年，我们拥有大量令人自豪的新项目，为正常运行和维护它们的更新承诺，和使用它们的充满活力的社区一起工作的愿望。仅在我们的 Github 账户上，我们现在已经有超过 90 个仓库，有超过 40,000 次的提交，一共被复制了 15,000 次。

年末是一个很好的机会来回顾我们投入精力的一些主要的领域，简要重述（不是详细的！）下我们工作的一些项目列表。

手机

我们最自豪的开源贡献常常是我们为了解决在 Facebook 遇到的规模和性能的挑战而开发的一些技术。

手机也不例外：Facebook 的手机应用已经是我们产品团队今年的一个重点，我们现在也还需要开发新的兼容性——在这种情况下，那些要迅速开发、编译、测试并发布我们手机应用的需求变得更加的高效。开源我们的工作，正如我们已经做的那样，是很自然的事情，这些工具已经成为我们手机开源作品集的核心。



比如，在四月份，我们发布了 Buck，我们的 Android 编译工具。开发者的效率对我们来说很重要，自然速度是 Buck 的首要工作：在 Facebook，我们发现它在编译我们的应用时，比 Ant 快了超过两倍。从一开始，这个项目就越来越强大，并被 Android 社区也包括一些有名的 Java 项目广泛应用。

对于 iOS，我们也有类似的瓶颈，发现我们需要加速和自动化 Xcode 的手动编译过程。结果就是非常流行的 xctool ——也在四月份发布——可以使工程师（更不说那些持续集成的机器）更加简单的编译和测试 iOS 和 Mac 项目。

我们在十月份举行的 Mobile @ Scale 上推出了 Rebound，一个 Android 的物理和动画库。Will Bailey 在这篇最近的博文中详细介绍了这个项目，我们相信对真实世界的物理建模是一个有力的方式，方便在应用中创建自然、可触的动画和交互效果。

最后，通过发布另一个手机工具来结束这一年，还有比这更好的方式吗？今天我们很激动的发布 Origami，一个 Quartz Composer 的工具集，允许设计者更快的编译和构建手机交互的原型。

Web

Web 技术也和 Facebook 相关，包括手机和桌面的客户端。在前段，我们开源的重点更多的在于支持在五月份的 JSConf 上发布的快速灵活的 JavaScript 库 React。

从那以后，React 库——和社区的热心者——有了一定的发展。现在在编译工具上有了大量的集成，服务器端的环境（像 node，Rails，和 Python）和其他客户端的库——也包括一些备受瞩目的外部的部署工具如 Khan Academy。团队维护了一个出色的社区概要博客，包括大量其他 React 项目的例子、演示和教程。

总之，我们也想帮助提高 JavaScript 语言和 web 应用的质量。例如，Regenerator，是用来转换 ECMAScript 6 的 yield 语法到现今的浏览器的一个转换器，Huxley，在夏天由 Instagram 团队构建的，已经快速成为 web 应用中一个非常流行的可视化正则测试工具。

数据

Facebook 与开源数据基础设施(open source data infrastructure)颇有渊源，回顾我们对 MySQL， Cassandra，Hadoop，Hive 和 Hbase 所做的贡献可以看出这一点。2013 年也不例外，仅仅在过去的几个月里，我们就上线了两个新的旗舰级数据项目。

首先是 Presto，一个新的分布式 SQL 查询引擎，被设计用于高性能分析我们用于运行交互式查询的 300PB 数据仓库。

我们在夏天的 Analytics @ Scale event 上公布了 Presto 并且在上个月于我们的 blog 上对它做了更多介绍。从那以后，我们激动的看到它被许多像 Airbnb

和 Dropbox 这样的公司采用，并且从许多热情的社区得到了贡献，包括 new clients，Ansible playbook 和 debian packaging。

第二个项目是 RocksDB，一个非常与众不同的数据基础设施(data infrastructure)：一个基于 LevelDB 的嵌入式 key-value 存储类库，并且为拥有多核 CPU 和快速闪存的环境做了优化。此外，在它上线的以后几个月里，它引起了广泛的兴趣，一些伟大的社区已经把它绑定到其他的语言上。

我们对数据基础设施(data infrastructure)的贡献和支持拓展得很好也已经超过了我们本身。我们对 Giraph 提供了强大的支持，比如，它已经被提高到可以支持 1 万亿边缘的图形结构。

基础设施项目

最后，但绝对不是最不重要的，基础设施项目在我们对开源项目的贡献中仍然是最为重要的一部分。

HHVM，the HipHop Virtual Machine，是目前为止我们的项目资产中最为显著并且是追随人数最多的项目，并且得到了大量的来自 PHP 生态系统的支持。2013 年有近 4000 次提交，并且在性能和第三方 php 框架的兼容性上有了长足进步，这对于社区的广泛采用来说是非常重要的。

这个团队刚刚从封闭中走出来并且在昨天分享了他们最新进度的消息。并且我们很激动的看到在持续不断的集成测试之后，VM 被集成到 Travis CI，并且为集成到其他流行的 web server 加入了 FastCGI 支持。

我们今年也为许多其他的显著的基础设施项目工作过。对于我们自己的项目，包括 pfff，我们的代码分析工具箱，libPhenom，一个高性能的事件框架，和 folly，我们流行的 C++类库。并且我们今年为 Mercurial，LLVM 和 GNU grep 提供了许多显著的贡献。

Open Compute 项目在 2013 年继续壮大，使用 new work on networking hardware，新的社区和基础管理，以及全年众多的 hackathons 和讨论会。Facebook 致力于支持项目建设并且它的目标是开发设计用于所有数据中心技术的开源服务器和数据中心 – 并且让我们期待下个月的 Open Compute Summit。

2013 年我们的开源项目...

当你在使用或者为一个开源项目做出贡献时，我们知道没有比看到它停滞不前更糟糕的了：bug 没有被修复，问题没有解决，pull requests 被忽略。关于 Facebook，我们的目标是，通过我们的投资组合，保持强有力的社区参与责任感。

我们已经采取了具体的措施来达到这个目标。例如：我们现在鼓励我们的工程师团队在使用或者开发一个开源项目的时候，首选 Github，Bitbucket 或者 Apache 软件基金会的项目作为源码的来源。

我们有一个新的工具链，使 repos 和我们内部系统同步，同步代码评审过程，任务跟踪等等。同时把我们最近检测的所有 repos 仪表化，保证他们保持

健康的发展：我们有内部的仪表盘来显示 commits 的数量，pull requests 的数量和每个团队 issues 的数量。这给我们提供了一个早期预警系统，提醒社区中有哪些被我们无意中忽略了。

同时我们很自豪的宣布，我们正在做的 Bountysource 项目帮我们社区解决了很多问题，其中包括鼓励大家参与各种开源项目，奖励那些参与开源项目的开发者。我们已经有了一系列的奖励计划，包括 HHVM 和 D。

就在上个月，我们宣布 Facebook Open Academy 已经把开源带到世界各地大学的计算机科学课程中。

在 2013 年，我们也同步开始了 "@ Scale" 工程活动的项目，这个项目旨在汇集开发人员讨论和分析各种技术大规模实施面临的挑战，分享他们的解决方案和相关的开源项目。今年的活动包括 Analytics @, Mobile @ 和 Data @ Scale，相关的视频已经放在了我们的 Facebook Developers channel。 敬请期待明年更多的 @ Scale 活动。

最后，欢迎访问我们新工程的网站 code.facebook.com！我们尽可能让您更方便的关注我们所有的活动详细信息，在 blog posts, events, videos, academic publications 和 open source projects 上都可以找到我们所有的工程项目和活动。订阅我们的 Facebook Page 和 open source news 可以获得更多的更新新闻。

...以及未来

Facebook 的著名格言：我们的开源项目仍然是只完成了 1%。

通过上面提及的所有主题，我们知道还有很多方面需要我们继续努力。我们很庆幸在我们众多的项目中有强大而热情的社区支持，给予我们强烈的责任感和动力。

不管你们是在移动端，web，数据亦或是基础设施社区，我们都很期待能继续与你们合作，2014 年再见！

原文：
http://www.oschina.net/translate/2013-a-year-of-open-source-at-facebook?utm_source=tuicool

趣文：如果老婆和女朋友她们是程序

去年，我的一位朋友和他的 GirlFriend 6.0 升级到 Wife 1.0 （也就是他们步入婚姻殿堂了）。婚后他发现，结婚就是只留给其他应用少量系统资源，自己却狂占内存的进程。老婆还要生成子进程（Child Processes），子进程会在将来消耗更多的资源。虽然产品说明书或手册里没有提及这种现象，但大家都知道这些都源于自然规律。

不只如此， Wife 1.0 在安装时设置了开机启动，监测所有系统活动。朋友发现许多应用，比如，扑克之夜、啤酒狂欢、午夜酒吧 已经无法在系统上运行了，每次运行，系统就会崩溃。

Wife 1.0 安装时并未给出提示，婚后却多了岳父岳母两个插件，系统性能看起来一天不如一天。

朋友希望 Wife 2.0 版拥有如下新特性：

- “不再提醒我”按钮；
- “最小化”按钮；
- 安装时增加新选项， Wife 2.0 可以在任意时间卸载，并且不会造成缓存和系统资源损失；
- 允许网络驱动使用混乱模式，以便系统硬件探测更多有用的功能。

我本人决定继续使用 GirlFriend 7.0 ，避免 Wife 1.0 带来的麻烦，即使如此，我依然麻烦重重。很显然， GirlFriend 7.0 不能安装在 GirlFriend 6.0 上面，必须先卸载 GirlFriend 6.0。这是一个众所周知的 bug，已经存在很长时间。很显然，上一版本的女朋友在 I/O 端口共享的问题上有冲突。我还以为她们已经修复好了这个愚蠢的 bug。更糟糕的是，GirlFriend 6.0 卸载时在系统中留下很多不良记录。还有，所有版本的女朋友都会不断弹出烦人的提示，提醒我升级成 Wife 1.0 的好处。

Wife 1.0 有一个没有记录在案的 bug。在 Wife 1.0 卸载之前，如果你试图安装 Mistress 1.1（情人 1.1） ， Wife 1.0 会在卸载之前删除所有财富文件，然后 Lover 1.1 就会因没有足够的系统资源而安装失败。

为了避免此种 bug，就要把 Mistress 1.1 安装在另外一个系统里，并且不要使用例如 Laplink 6.0 这样的通讯软件。同时要小心那些已知携带病毒感会感染 Wife 1.0 的相似共享软件。另外一个解决办法是匿名，通过 UseNet 提供的网络运行 Mistress 1.1，再次提醒，小心那些能从 UseNet 下载东西的病毒。

技术支持的建议：

男同胞抱怨的很多常见问题，大多是源于误解。很多人把 GirlFriend 6.0 升级为 Wife 1.0 ，是因为 Wife 1.0 是最主要的实用工具和娱乐程序。实际上，Wife 1.0 操作系统被原作者设计为可以做任何事情。

从 Wife 1.0 退回 GirlFriend 6.0 是不太可能的，系统内的隐藏文件会使 GirlFriend 6.0 模仿 Wife 1.0，所以一切都是徒劳。一旦安装，就不可能从系统内卸载、删除、粉碎程序。从最初的设计上就屏蔽掉此功能。

一些人安装了 GirlFriend 7.0 或者 Wife 2.0 ，但问题比最初系统还糟糕。看看手册里的 “生活费/子女赡养费“ 警告，我建议你在 Wife 1.0 下处理问题。我建议你安装后台应用程序“c:\遵命亲爱的“ 以减缓软件增大。

本人在安装完 Wife 1.0 后，建议你认真阅读“常见关系故障”章节（General Partnership Faults - GPF）。你必须为所有可能发生的过失和问题承担责任，不管其原因。最有用的是在命令行里输入 c:\我道歉。

要避免滥用“c:\遵命亲爱的“，因为有可能在系统恢复正常之前，你还得用 “c:\我道歉“ 命令。只有把所有责任都揽在自己身上，系统才能平稳运转。

Wife 1.0 是一个很棒的程序， 但是维护成本很高。为提高 Wife 1.0 的性能，再买些辅助软件吧。我推荐 “鲜花 3.1”和“钻石 2K “。千万不要安装”短裙秘书 3.3 “，它不支持 Wife 1.0，而且会造成操作系统不可挽回的损失。

原文：http://blog.jobbole.com/53403/?utm_source=tuicool

Reddit 帝国建立在一个有瑕疵的算法之上

Reddit 的源码中存在一个 bug。这个 bug 目前还存在他们的平台产品之中，并且已经存在了多年。这个 bug 与应用于整个站点最重要的算法之一有关——针对“热点”链接受欢迎度的排序算法。这个 bug 也导致了真实显见的负面影响，并且这个问题多次报告给 Reddit 的技术组，但是一直没有被修复。

缺陷

Reddit 需要判断当前哪篇文章比较热门。新文章比旧文章更热门一些，拥有更多正面投票的文章比有很少投票的文章更好，而大部分投负面票的文章则垫底。这些规则比较容易计算。针对发布时间和投票可以确定一个确切的数值，然后与一个常量系数，就能计算出每篇文章的的受欢迎程度[1]。

魔鬼在于细节，根据 GitHub 中的源码，目前的实现是这样的。

```
seconds = date - 1134028003
```

seconds 是根据时间变化的变量，基于 Unix 时间戳。这样做合情理，时间总是累加的，所以每一个新文章发布都较之前发布的文章在时间上拥有更高的分数值。

```
s = score(ups, downs)
order = log10(max(abs(s), 1))
if s > 0:
        sign = 1
elif s < 0:
        sign = -1
else:
        sign = 0
```

投票部分计算根据公式有两个部分，sign 变量简单地记录总投票数是正面还是反面的。如果文章收到的正面评分比负面评分更多，sign 就是 1；如果负面评分更多，sign 的值就是-1.其他变量，order 是投票分数绝对值的取 $\log10^2$ 对数。

真正的问题是，和许多问题一样，从两个角色的交换。

```
return round(order + sign * seconds / 45000, 7)
```

这里我们计算得到了最终的分数。Seconds 是一个很大的正值，而 order 总是返回为正——由于这里使用了的绝对值，所以，尽管如一个负数-389 也会得到和 order 值为 389 一样的结果，我们需要通过 sign 来调整我们的结果，这样，负面意见的文章会被排在后面。但是这里的代码使用了 sign 乘于时间 seconds，而不是 sign 乘于 order。

而对于正面评价的文章,则没有影响,由于符号位为 1,所以 order 和 seconds 相加，计算没有问题。

对于负面评价文章又发生了什么呢？sign 值为-1,所以值比较大的 seconds 变量会变为负值，而使用一个正面评价的 order 值与之相加，因此会导致几个问题。

假设两篇文章，相距 5 秒发布。每一个都收到两个父母评价，seconds 对于新文章值更大一些，但是因为 sign 为负值，新的文章评分反而比旧的文章低。

假定还有两篇文章同时发布，一篇文章收到了 10 个差评，另外一篇文章收到五个差评。由于 seconds 一样，符号值 sign 都是-1，但是得到-10 评价的文章 order 值更高。所以，-10 评分的文章反而排序在得到-5 评分的文章前面，尽管人们讨厌它两倍。

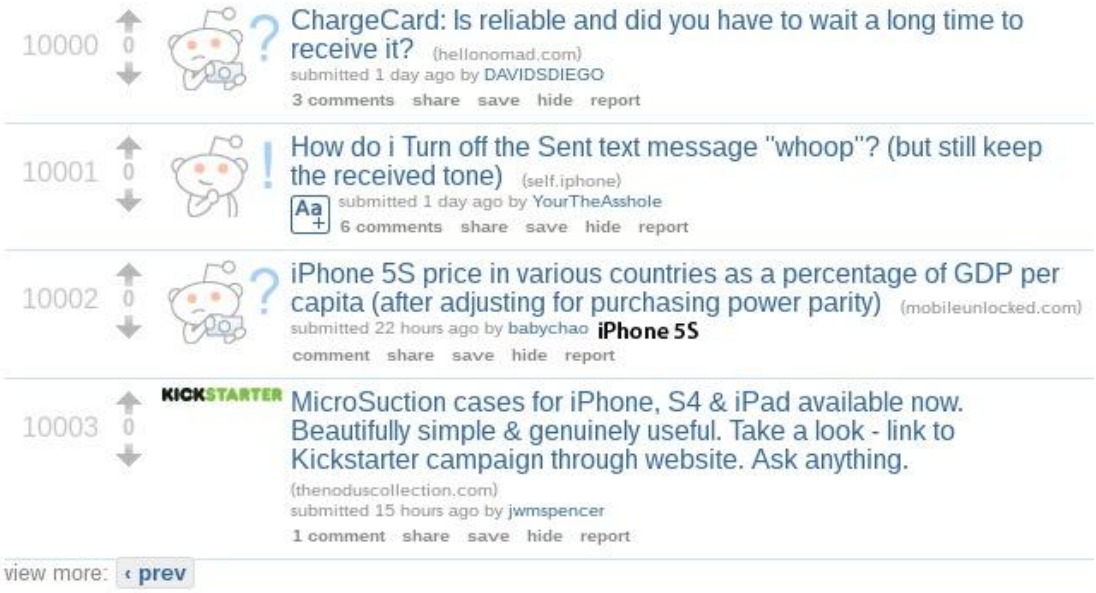
现在假定一篇文章在一年前发布，另外一篇文章刚刚才发布。去年发布的文章有两个正面评分，今天刚发布的文章有两个差评。这里有点不同——今天发布的文章可能得到了差评开始，不过也可能接下来收到好评。不过，在当前的实现[3]中，由于今天文章有两个差评，从而导致其热度评分反而比去年的文章更低。

后续

这并不是一个假设的漏洞，我好奇地去看了一些 Reddit 的公共库的代码是否也应用在他们当前的产品中，我找到近期发布的一些不太活跃的文章，给它差

评使得它的总分是负分。就是这样的文章，不但跌出了第一页（这一页当中还有几个月前的文章），这个是热度排序算法的结果。我觉得这个结果比较糟糕，又移除了我的差评，但是这个文章并没有恢复到排名之中[4]。

事实上，通过操作查询字符串，你可以发现一个奇怪的陷阱，糟糕的文章在莫名的角落缓慢地腐化[5]（译者：我猜作者的意思应该是让人讨厌的文章总数显示在显眼的地方）。下面的截图来自 iphone 版的 subReddit 这些不幸的文章。



这些文章是一些伤心的、害怕、孤独的文章。但是在显著位置，并且按时间顺序排列，就如预料的一样。

这个缺陷为一些故意破坏这个系统的人提供了一扇门。假设有一个 subReddit，/r/BirdPics，投票给一些鸟类图片[6]。攻击者不喜欢海雀，并且希望把所有的海雀图片都驱逐出首页，这个攻击者可以对每一张海雀图片都给差评，不过这也可能被那些喜欢海雀的人给好评而消弱。平均来说，任何时刻查看首页的人的大概是 350 人，为了阻止这种恶意差评需要许多的评价较量。

不过，攻击者可以小心留意那些最近发布的图片，这个时候一旦有海雀图片被发布，攻击者立即给他一个差评。如果攻击者先投票，那么这篇文章总分就是一个差评并且会被逐出首页，在首页上再也看不到这篇文章。攻击者唯一需要担心的是有人会去查看最新排序文章列表，这里投票并不影响排序。我们假定 350 人中只有 10 人会去查看最新文章列表，因此攻击者只需用使用 10 个虚假账号来应对这些人，而不是需要近 300 个账号来对付首页上的所有人。就这样，攻击者可以清除 subreddit 上的所有海雀图片，这些海雀就再也没人注意。

补救措施

我不是第一个发现这个问题的人，Jonathan Rochkind 在他的发表的一篇言辞详尽的话题中谈及了该问题。不过一位 Reddit 的开发者告诉他，他弄错了，当前的算法没有问题。

我在开发者社区发布了一篇请求修复该 bug 的文章，也被一个开发者告知，就是这样设计的。我不懂，也没有得到一个令人满意的解释，在何种意义上这种荒谬的行为将是“就是这样设计”的。但很明显，Reddit 对解决这个问题不感兴趣，这一行为可能会持续许多年。

结局

程序员往往围绕着规则的一致性寻找公平正义。这也是为什么我们中的许多人发现世俗领域关系或政治如此棘手的，以及为什么我们中的许多人第一时间被计算机科学吸引。在计算机世界，一切都是严格确定的。如果发生了一些未定义的事情，那也只是因为我们对系统的理解不完整[7]。对于正确性，capital-R 就是一个可理解的系统并且精确执行。

我们拥有这样的世界观，而一个明显的 bug 被散播看起来是不公正的。我和其他发现这个问题的开发者看起来对这个算法的理解比那些 Reddit 回应我们的员工更深入一些。对于这个没有被修补令人惊讶且违反直觉的行为，我们肯定是对的。我们是对的，而 Reddit 搞错了。因为 Reddit 是一个流行站点，并且拥有大量的用户，以及大量的现金。这些都建立在一个明显错误的组件之上。

这里的道德是什么？也许就是一个没有有效测试使得这个系统不能被理解，或者最终变为一个“可以工作，但是停止问问题”的系统。也许追求完美是好的一面的敌人，更糟反而更好[8]，这样的分歧使我远离喋喋不休的争吵 9。也许好产品和好的技术实现存在一定的距离，而硬性的数据总是能保证正面的体验。

也许这没有道德，是 Reddit 搞砸了。这原本会伤及它自身，但是没有，也可能不会。他们错了，但是没有导致实际的错误，因为没有一个大写 W Wrong。我们都应当编写有责任的代码，这里没有一个数学上帝可以有一天评判 Reddit 在做算法时的罪行。世界就是一个有缺陷的世界，曾经是，也一直都会如此。

脚注：

1. 这个想法简单而有效，你可以根据这个算法构建不同的站点，但是使用不同的常量。如果你想要一个站点展现一些很旧的文章，可以将时间变量的权重调低。需要一个 twitter 站点，将 vote 变量的权重设置为 0.

2. 这个对数计算使得对 Reddit 的投票在权重上相差悬殊。1 个投票和 11 个投票的差距比 10001 个投票和 10011 个投票的差距要大得多。

3. 这个计算使得时间 seconds 的值很大，相对于 order 始终拥有更大的值。在 Reddit 的实现中是这样。

4. 通过测试，我无法确定是投票统计的波动导致评分变化究竟发生了什么。我现在知道了，这个是一个投票混淆算法，有点类似反垃圾邮件功能。

5. 我不能提供关于这个缺陷的持久化链接，这个索引好像一天后就会失效，不过这也容易找到。首先，查找最近负面评分的文章，记录下它的 ID 值，这个可以从 URL 中找到，比如：
http://www.reddit.com/r/birdpics/comments/1s33tt/fear_the_shrike/，我们可以得到 ID 值 1s33tt。然后，插入下面的链接，取代必要的部分：
http://www.reddit.com/r/SUBREDDIT/?count=9999&after=t3_ID，。我们的 URL

会变成 http://www.reddit.com/r/birdpics/?count=9999&after=t3_1s33tt。这里 ID 值被 t3_前缀展开，然后你可以随意改变 count 值，这个值可以手动调节。

6. 这个当然存在，见：链接。

7. 我怀疑，这也是为什么海森伯格测不准是计算机科学家最害怕也最讨厌的问题了。参见：释放 Zalgo。

8. “糟糕反而更好”来自于 Richard Gabriel 关于研究 C 的兴起和 Lisp 的衰微的开创性文章。这篇文章和后续的一些续篇是关于计算科学迄今最好的一些文章。

9. 你能猜出这些类比我是针对哪一个错误的吗?

原文：http://blog.jobbole.com/53406/?utm_source=tuicool

项目开发中，你会倾向于质量还是速度？

在项目开发中，你会倾向于质量还是速度？当然，两者都很重要，理想的情况是，在规定的时间段内高质量的完成所有的东西。**但是往往现实是比较残酷的，很少会给你这种机会，使得这两者不能兼得。**

在快节奏的开发工作中，你必须争分夺秒，以在最后期限之前项目能够如期交付。但时间一紧，就容易忽视代码的质量和规范，或者不去写测试用例。反过来，如果太过追求项目的质量，则会拖延进度。

当客户需要你尽快交出成品的时候，要么使产品中的一部分整洁美观的，要么使产品是完整的，但有些部分不尽如人意。**那么如何在这两者之间找到一个平衡点呢？来看开发者 Matt Aebersold 的建议。**

项目开始时就注重代码质量会加快进度

好的代码是一种“艺术”，优雅、整洁、易于阅读、团队协作也比较容易。这是我们应该在每一天都要努力的方向。如果项目一开始就注重代码的质量，那么在项目中后期，事情将会变得简单。比如，创建一个 JS 文件来保存所有配置级别的变量，那么在后期需要调整一些类似于动画速度和延迟时间方面的东西时，就会变得易如反掌。

按计划完成，留出改进时间

在开发者关于这方面的探讨中，速度往往容易引发争论。我支持快速开发的原因有很多，其中最主要的是要按时或更早地将任务完成，然后留给改进工作更多的时间，这可以使得产品人员和客户都高兴。

有时简单是最合适的方式

毫无疑问，创建一个框架可以加快开发速度，但不是一切都适合使用框架。比如一个非常简单的需求，可能只需要一个简单的标签或这脚本就能够解决问题，而你非要去花费时间构建一个创新的方式或工作流程，这是毫无必要的。

开发项目中，从大的框架到小的脚本，都可以用在项目中，但是一个优秀的开发会去挑选什么才是最适合该项目的东西，而不是在所有情况下都使用最复杂的技术。

找出项目中什么才是最重要的

在项目开发过程中，你应该考虑大部分时间应该花在什么地方。例如，如果该网站不需要复杂的 JavaScript，那么就不要添加一些 JavaScript 框架和模块，因为这需要时间和精力。相反，一个简单的脚本文件，甚至是一些内联 JavaScript 代码就会工作得很好。这样一来，你可以花更多的时间在网站上的其余部分。

如果项目是你个人的，那就花费所有时间确保把每一行代码都写好，将其优化到最简洁的形式。如果项目必须在某个时间内完成，那么就选择一条能到达终点的最短路径。我在过去 5 年内，95%的情况都是后者，我也在努力在最短的时间内完成高质量的工作。

原文：http://www.iteye.com/news/28585?utm_source=tuicool